On the Synthesis and Optimization of MVL Functions

by

Muhammad Nayyar Hasan

A Thesis Presented to the

FACULTY OF THE COLLEGE OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In

COMPUTER ENGINEERING

June, 1995

On the Synthesis and Optimization of MVL Functions

BY

Muhammad Nayyar Hasan

A Thesis Presented to the

FACULTY OF THE COLLEGE OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE
In

Computer Engineering

June 1995

KING FAHD UNIVERSITY OF PETROLEUM AND MINERALS

DHAHRAN, SAUDI ARABIA

COLLEGE OF GRADUATE STUDIES

This thesis, written by

Muhammad Nayyar Hasan

under the direction of his Thesis Advisor, and approved by his Thesis committee, has been presented to and accepted by the Dean, College of Graduate Studies, in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER ENGINEERING

Thesis Committee :

Prof. Dr. Mostafa Abd-El-Barr (Chairman)

Dr. Sadiq M. Sait (Member)

Dr. Habib Youssef (Member)

Department Chairman

Dr. Ala H. Rabeh
Dean, College of Graduate Studies

Date: 4/7/95

On the Synthesis and Optimization of Multi-Valued Logic (MVL) Functions

MS Thesis

Muhammad Nayyar Hasan

June, 1995

Dedicated to

my mother (late)

and my father

whose prayers, guidance, and inspiration led to

this accomplishment

Acknowledgment

In the name of Allah, Most Gracious, Most Merciful. Read in the name of thy Lord and Cherisher, Who created. Created man from a {*leech-like* } clot. Read and thy Lord is Most Bountiful. He Who taught {the use of} the pen. Taught man that which he knew not. Nay, but man doth transgress all bounds. In that he looketh upon himself as self-sufficient. Verily, to thy Lord is the return {of all}.

(The Holy Quran, Surah 96)

First and foremost, all praise to Allah, *subhanahu-wa-ta'ala*, the Almighty, Who gave me an opportunity, courage and patience to carry out this work. I feel priviledged to glorify His name in the sincerest way through this small accomplishment. I seek His mercy, favor, and forgiveness. And I ask Him to accept my little effort. May He, *subhanahu-wa-ta-Aaala*, guide us and the whole humanity to the right path *(Aameen)*.

Acknowledgement is due to King Fahd University of Petroleum & Minerals for providing support to this research work.

I am indebted to my thesis chairman, Dr. Mostafa Abd-El- Barr for his help and advice. I acknowledge him for his valuable time, encouragement and guidance especially during the early stages of this work and my MS studies. Working with him was indeed a learning experience.

I am grateful to my thesis comittee member, Dr. Sadiq M. Sait for his deep interest, constructive criticism and stimulating discussions during the course of this work. Thanks are also due to the my thesis committee member, Dr. Habib Youssef for his comments and critical review of the thesis.

I am thankful to the department chairman, Dr. Samir Abdul Jauwad and other faculty members for their cooperation.

I am thankful to my fellow graduate students and all my friends on the campus especially Asim, Amir, Asif, Ikram, Salman, Rashid, Sabih and Zaka who provided a wonderful company.

Last but not the least, thanks are due to the members of my family for their emotional and moral support throughout my academic career. No personal development could ever take place without the proper guidance of parents. This work is dedicated to my parents for taking pains to fulfill my academic pursuits and shaping my personality. They taught me the fundamental concept of life, "Tough

times never last, tough people do". I acknowledge with gratitude, the affection and encouragement of my cousin Shamim that helped me overcome homesickness and concentrate on my work. I also acknowledge the support of my nephew and niece who made my stay in Saudi Arabia a enjoyable one.

Contents

List of Tables

List of Figures

Abstract

Name: Muhammad Nayyar Hasan

Title: On the Synthesis and Optimization of MVL functions

Major Field: Computer Engineering

Date of Degree: June, 1995

Multi-valued Logic Circuits (MVL) allow signals to have more than two levels. Some of the advantages of MVL circuits are increased functional density and less requirement of interconnection wiring which enable designers to utilize the chip area more efficiently.

In this thesis, a number of decomposition based mapping techniques are proposed for the synthesis of MVL functions. These decomposition based techniques use a matching count matrix for input-output pairing that result in less number of switching operations. In addition an optimal output phase cyclic assignment technique is presented to reduce the number of implicants required to realize a given MVL function.

Programmable Logic Array (PLA) is important sub-systems in the design of digital integrated circuits. This is due to its regular structure, ease of testability and automatic synthesis. PLAs can also be used to implement complex MVL circuits. In this thesis some MVL-PLA structures are proposed which use current-mode CMOS building blocks such as min, cycle, tsum etc. The proposed architectures are compared with each other in terms of the functional coverage. Furthermore, the simulations of current-mode CMOS building blocks used in the proposed PLA architectures have been carried out to verify their functionality.

Master of Science Degree
King Fahd University of Petroleum and Minerals
Dhahran, Saudi Arabia
June, 1995

خلاصة الرسالة

اسم الطالـــب : محمد نيـر حسن

عنوان الرسالة : التجميع والتمثيل الأمثلي لتوابع المنطق المتعدد القيم .

التخصـــــص : هندسة الحاسب الآلي .

تاريخ الشهادة : يونيو ١٩٩٥ م

تسمح الدوائر ذات المنطق المتعدد القيم للإشارات بأن يكون لها أكثر من مستويين آثنين . ومن بعض مميزات الدوائر ذات المنطق المتعدد القيم هي تزايد الكثافة الوظيفية وتقليل الحاجة الى الاسلاك بين الوصلات مما يسمح للمصممين باستخدام مساحة الشريحة بشكل أكثر فعالية .

في هذه الأطروحة يتم اقتراح عدة اساليب تحويل تعتمد على التجزيء الى عناصر لتجميع التوابع ذات المنطق المتعدد القيم . وهذه الأساليب تستخدم مصفوفة عداد مطابقة لمزاوجة الدخل - الخرج مما يؤدي الى التقليل من عمليات التبديل وبالاضافة الى ذلك ، فإنه يتم عرض اسلوب تعيين أمثلي تكراري لحالات الخرج الثنائية لتقليل عدد العناصر الحسابية للتابع المنطقي ، وذلك لتحقيق تابع معين ذي منطق متعدد القيم .

ويُعتبر المصفوفات المنطقية المبرمجة (PLA) أجزاء مهمة من الأنظمة في تصميم الدوائر الرقمية المتكاملة. إلى بنيتها المنتظمة ، سهولة الفحص وامكانية التجميع المؤقت . ويمكن استخدام المصفوفات المنطقية المبرمجة (PLAS) لتنفيذ دوائر المنطق المتعدد القيم المعقّدة .

وفي هذه الأطروحة ، يتم اقتراح بعض تراكيب المصفوفات المنطقية المبرمجة ذات المنطق المتعدد القيم والتي تستخدم وحدات تكوني من نوع CMOS ذات صيغة التيار . ويتم مقارنة التراكيب المقترحة مع بعضها البعض تبعاً لعدد التوابع التي يغطيها كل تركيب . ولقد تم اجراء عملية محاكاة لكيفية عمل تراكيب CMOS المستخدمة في بناء المصفوفة المنطقية المبرمجة المقترحة للتحقق من عملها على الشكل المطلوب .

درجة الماجستير في العلوم

جامعة الملك فهد للبترول والمعادن

الظهران ، المملكة العربية السعودية

يونيو ١٩٩٥ م

Chapter 1

Introduction

1.1 Overview

Two-valued (binary) signals currently dominate the field of digital computing. However, during the last few decades design using Multiple-Valued Logic (MVL) has been gaining considerable attention [1]. MVL allows circuits to have more than two levels. With the advancement of VLSI (Very Large Scale Integration) in IC (Integrated Circuit) technology, the need for reduction in interconnections between active devices both on and between chips is becoming more and more pressing. It is generally accepted that routing of interconnections on a chip is a difficult problem. It has been estimated that in conventional binary integrated circuits about 70% of chip area is used for interconnection wiring, 20% for passive isolation and about 10% for active devices [2]. One can then conclude that even if all dimensions

of active elements were reduced to zero, the packing density of a binary logic chip would hardly change. Similarly, the difficulties of bringing an increasing number of connections off-chip is promoting a new consideration to packaging concepts [3].

In order to reduce the interconnection wiring, the information content of each connection and/or chip pin must be increased either by time-multiplexing or by level-multiplexing [4]. The first approach is mainly limited to inter-chip pin connections and the second technique can be applied to intra-chip circuitry. In time-multiplexing, a pin of an IC is used to carry different signals at different times whereas level-multiplexing allows an interconnection to carry an r-valued signal (r>2).

The pin limitation problem in microprocessor dual-in-line package is solved by time-multiplexing. However, this method is not efficient as it results in overall speed performance degradation. In addition, multiplexing of chip input and output signals also requires more off-chip and on-chip interfacing circuitry. Another solution to the pin-out limitation can be provided by developing specific packaging techniques with more than 200 pins (e.g., regular pin-grid-array package)[5]. Although these packaging techniques solve the pin-out problem at the chip level, they cause an interconnection problem at the board level. So, one can conclude that neither time-multiplexing nor new packaging techniques provide an absolute solution to the pin-out problems. The introduction of multivalued logic circuits could alleviate these problems.

Circuits which allow more than two logic levels are called Multiple Valued Logic

Table 1.1: Truth table for an example 4-valued 2-variable function.

input		output
x_1	x_2	$f(x_1, x_2)$
0	0	0
0	1	1
0	2	2
0	3	3
1	0	1
1	1	2
1	2	3
1	3	0
2	0	2
2	1	3
2	2	0
2	3	1
3	0	3
3	1	0
3	2	1
3	3	2

(MVL) circuits. Similarly, those functions which use more than two logic levels for their inputs and outputs are called multi-valued functions. As an example, consider the truth table of a 4-valued 2-variable function shown in Table 1.1. Both the input variables (x_1 and x_2) and the output $f(x_1, x_2)$ can have 4 logic values. When x_1 and x_2 are 0 then $f(x_1, x_2)$ is 0. Similarly, when x_1 is 1 and x_2 is 1 $f(x_1, x_2)$ is 2. The truth table of Table 1.1 can also be represented in map form as shown in Figure 1.1. For the sake of clarity, zeros are not shown for $f(x_1, x_2)$.

It has been shown mathematically in [3] that higher radix numbers are more economical in terms of the number of digits required to express them. Smaller radix

x2 \ x1	0	1	2	3
0		1	2	3
1	1	2	3	
2	2	3		1
3	3		1	2

Figure 1.1: Map representation of example 4-valued 2-variable function.

numbers require more digits to express. The number of digits required to express a given number N is given by $d = \frac{\log_2 N}{\log_2 R}$, where R is the radix and d is the number of digits. Assuming that the cost or complexity C of hardware is proportional to the digit capacity $(R \times d)$ then

$$C = k(R \times d) \tag{1.1}$$

$$= k \times R[\frac{\log_2 N}{\log_2 R}] \tag{1.2}$$

where k is some constant. Differentiating the above equation with respect to R shows that for minimum cost C, R should be equal to $e = 2.718$. Since for practical purpose R must be an integer, therefore according to the above simple analysis $R = 3$ would be more economical than $R = 2$ (binary). Also, if the circuit cost and complexity C for processing one signal line remains constant irrespective of radix, then the total system cost C is proportional to d.

$$C = k \times d = k[\frac{\log_2 N}{log_2 R}] \tag{1.3}$$

This equation shows that the cost gradually decreases with the increase in the radix R. These results are shown in Table 1.2.

Table 1.2: Theoretical variations of total system cost and interconnect, assuming binary system value of 100. [3]

Radix R	Cost C, assuming proportional to $R \times d$	Cost C, assuming proportional to d	Number of interconnect lines required compared to 100 binary lines
1	∞	∞	∞
2	100	100	100
3	95	63.1	64
4	100	50	50
5	107.9	43.1	44
10	150.5	30.1	31

This result can be extended to higher-valued logic. The table shows that for a given number of interconnections, it is possible to transmit more information using MVL signals as compared to binary logic. For example, a 4-valued signal on a single wire carries twice the information that can be carried using a binary signal. Also, many logic and arithematic functions have been shown to be efficiently implemented with MVL. These circuits required fewer operations, less number of gates, transistors and signal lines [6].

Despite the above mentioned advantages, there are some significant problems associated with the design and development of MVL circuits. The most common one is the reduced noise margins and tighter logic level tolerances. Since noise does not possess much problem within the chip as outside, it can be argued that noise

margin problem can be solved by binary interfacing to the outside world. A second problem with MVL might arise due to increased chip area resulting in excessive power dissipation [7]. Yet a third problem with MVL might occur due to increased circuit complexity and functionality resulting in a slower switching speed [6]. There is another problem, which is not directly related to MVL circuit itself, i.e., the lack of an established market for MVL components. Currently, the computer world is binary and this is not likely to change in the near future. The availability of large variety of binary circuits does not encourage logic designers to explore MVL alternatives [4].

1.1.1 Applications of MVL

Despite some of the disadvantages mentioned above, it is expected that the overall system characteristics can be improved by the use of MVL circuits. With the advancement of processing technologies many MVL circuits are becoming available for logic designers [5]. There is a growing number of commercially available ICs exploiting the advantages of MVL circuits. For example, in 1980, Intel announced 8087 math co-processor and iAPX-432 microprocessor. Both processors used four valued read only memory (ROM) for the microprogram control store [8, 9, 10]. Each cell of the ROM is capable of storing four-valued quantity, which may be viewed as an encoding of two binary values. These provide an approximately 30% ROM area saving compared to a binary ROM. In 8087 ROM, it also results in 8% reduction in

die area with a corresponding increase in die-per-wafer yield. There is also no speed penalty because MVL ROM was fast enough to respond within the time allocated for ROM data lookup [6, 11]. Following the two above described products, Intel has used the 2 bit/cell ROM in a number of products, among them the 82586-the Local Communication Controller which implements the IEEE 802.3 CSMA/CD protocol. Motorolla has used a similar approach to design their MCM65256- a CMOS 256K ROM. There is a 40% saving in memory array area and an overall silicon saving of 25% [11, 12].

Multi-valued signaling has also been applied in point-to-point link communications. It has been shown in [8, 13] that current mode signaling on point-to-point links is relatively straight forward. Such a link used I^2L quaternary logic.

In 1988, a very high speed CMOS MVL multiplier has been designed by Kawahito et.al [14]. The speed of this 32×32 multiplier is comparable to that of the fastest known binary valued multiplier, but it requires only half the chip area and half the power dissipation. This 32-bit multiplier uses signed-digit number system($\pm 2, \pm 1, 0$) and it also limits the propagation of carrys which is a major source of delay in binary valued multipliers. This results in the reduction of interconnection, allowing a much more compact circuit.

MVL circuits have also been used to augment binary circuits. Sasao [15] applied the multiple-valued logic theory to the design of binary PLAs. MVL circuits can also allow simple tests for stuck-at-faults in binary circuits [8, 16]. It is conceivable that

future IC's may be designed for normal operation in binary mode, and be switched to MVL mode for testing. Multiple valued logic has also been used in the design and analysis of binary systems. For example it has been used in the minimization of binary PLAs [17].

Researchers are also looking at the possibilities of introducing the MVL techniques in Field Programmable Gate Arrays (FPGAs) [18]. In MVL-FPGA the programming switch and interconnet lines can carry $\log_2 r$ times the information carried by the binary link. It might also be possible that MV logic blocks have more functionality per unit than their binary counterparts.

MVL has considerable scope for use in VLSI, if it is easily interfaced with binary inputs. There are some criteria for the choice of the value of radix R for MVL circuits. Since the present electronic world is dominated by binary, it is advantageous to choose radix to be a power of 2 [19]. This allows simple conversion to and from binary, thus allowing easy utilization of the wide variety of binary components. This binary conversion is also very efficient where no information is lost or unused. For example, ternary logic circuits can be interfaced with binary logic circuits by assigning two bits to each ternary digit. Using this method, one of the combination of two binary bits is not used. The MVL radix must be a power of 2 in order to utilize the functional density efficiently and for adequate interfacing with the binary world. The choice of higher radix value to encode more information is not efficient due to tighter tolerance and reduced noise margins. The most practical choice with

the currently available technologies seems to be $R = 4$.

1.1.2 Role of binary curcuits in MVL

Researchers have different opinions over whether to use fully MVL circuits or mixed-radix logic circuits. Some researchers [1, 5, 19] are of the opinion that MVL has to co-exist with binary logic whereas some others [20] argued in favour of fully MVL circuits and viewed that future chips will use only MVL signals. Vranesic [19] has proposed two bus structures for mixed-radix logic. The first one uses binary signalling on the bus (Figure 1.2) and the other one uses multivalued signalling (Figure 1.3). Both structures have conversion circuitry i.e *encoder* and *decoder* in order to allow MVL elements to communicate with binary counterparts and vice versa. The conversion circuitry may either be a part of the MVL chips (as indicated by the dashed line in Figure 1.2), or on separate chips. The main advantage of using the 2-valued bus system is that it can take advantage of the existing binary logic circuits. It is not advantageous to use MVL bus structure until MVL parts are in abundance.

Therefore, as long as the number of MVL parts is small as compared to the total number in the system, it is preferred to use binary bus structure.

The mixed-radix design (i.e binary and MVL circuits on the same chip or in the same system) is cost effective if the size of the conversion circuitry is small and if the MVL circuits are implemented by the same fabrication processes used for the

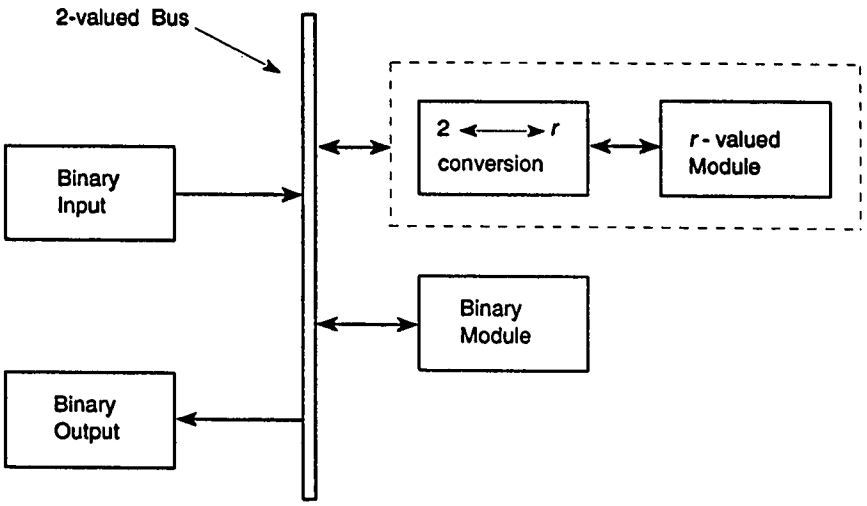


Figure 1.2: Two-valued bus structure [19].

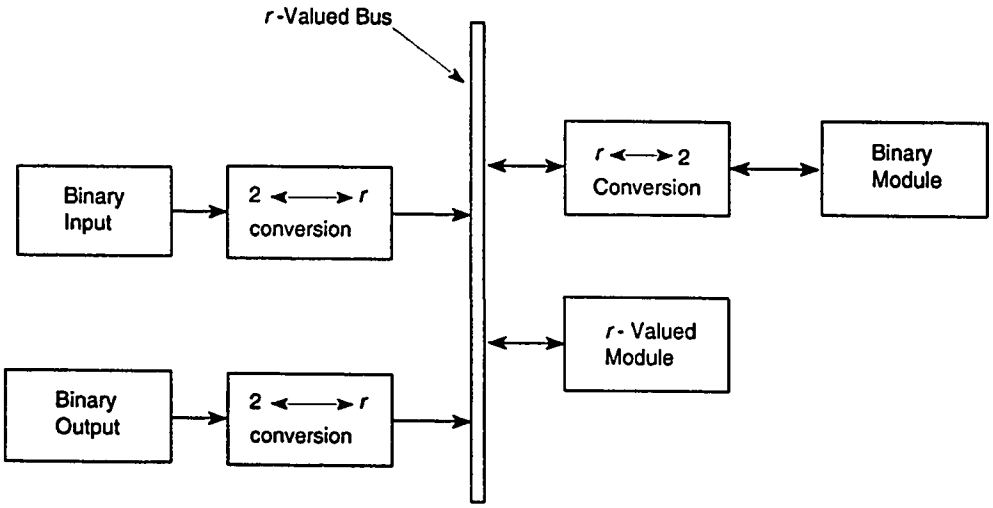


Figure 1.3: Two-valued bus structure [19].

binary logic circuits.

Several designs of quaternary encoder/decoder have been proposed in the past with different technologies. In 1984, Current [6] has implemented a quaternary encoding/decoding circuits using current-mode CMOS technique. In this design MVL signals were represented by current levels. Similar encoder/decoder has been designed by Butler and Kerkhoff [21] using charged-coupled devices (CCD) technology, where the MVL signals were represented as an amount of charge. Abd-El-Barr [22] proposed a different approach for 4-valued encoder/decoder circuits using CMOS technology. He has combined amplitude encoding with time division as two attributes to represent MVL values. The four logic values were defined by using two non-zero voltage levels and two pulse durations.

These encoder/decoder designs support [6, 21, 22] mixed-radix design but it has some drawbacks too. One drawback is that encoder/decoder requires extra chip area. Other drawback is degraded performance of the circuit in terms of speed which is due to conversion requirement.

Fully MVL approach allows circuits to take advantage of the reduced interconnection wiring and increased functional density. Also these circuits do not require any conversion circuitry. Heung and Mouftah [20] have implemented some circuits using 3-valued logic with enhancement and depletion type CMOS transistors. Abd-El-Barr et.al. [23] have used a totally MVL approach for synthesis of MVL circuits using CCD (charge-coupled device) technology. However, fully MVL approach is

also not without drawbacks. Reduced noise margin and tighter tolerances for high-radix are the problems faced by the fully MVL circuits. Presently, both approaches, i.e, fully MVL and mixed-radix realization are being used for circuit implementation. It is hoped that mixed-radix approach will continue to be used most often because of the availability of a large number of binary parts and also due to the fact that most logic designers are familiar with binary logic. However, it is anticipated that MVL will have considerable scope for use in VLSI in the near future due to advancement in processing technologies.

1.1.3 Research Motivation

Generally, the cost of multivalued circuits is greater than that of 2-valued circuits. Thus it is particularly important to have effective means of reducing the total number of circuits in such networks [24]. Some simplification techniques for minimizing the number of MVL circuit elements have been proposed in the past [24, 25]. In [24], a functional tranformation technique has been proposed which provide a powerful means for simplification of multi-valued switching functions resulting in reduced cost implementation. Similarly, Abdel-Hamid and Abd-El-Barr have presented a mapping technique for the synthesis of binary and MVL functions resulting in reduced circuit implementation [25]. Their approach considers only output phase technique. In this thesis we are proposing some other approaches for the synthesis of MVL functions based on a mapping technique.

Programmable Logic Arrays (PLAs) are important sub-systems in the design of digital integrated circuits. This is due to their regular structure, ease of testability and automatic synthesis. PLAs can also be used to implement complex MVL circuits [26]. In the past, several techniques for optimizing the structure of a PLA have been proposed. The goal of the optimization is to minimize the area occupied by the PLA. The area of the PLA is proportional to the number of product terms realized by the PLA. Generally speaking, two techniques are used for optimization of PLA i.e. *input variable assignment* and *output phase assignment* [27]. Sasao has used both techniques in his proposed MVL-PLAs [26]. He has proposed three types of MVL-PLAs. The Type 1 PLA realizes the function directly in the form of the MAX of MIN of literal functions and constants. In Type 2 PLAs, the body of the PLA is binary and the output is encoded as a multiple-valued logic value. Type 3 PLAs are the same as Type 2 PLAs except 2-bit decoders and a permutation network is used on the input. Trimulai and Butler [28] show two types of programmable logic arrays. Both are based on CCD technology and use universal literals of the input variables to generate product terms. These are then combined using either the SUM or the MAX operation to obtain the desired logic function. The architecture proposed by Dueck and Miller [29] is almost the same as the one proposed by Trimulai and Butler with the exception that Dueck and Miller proposed the use of MODSUM operation for combining the product terms. Abd-El-Barr [30] has proposed programmable logic array structures for the implementation of multi-valued multi-threshold (MVMT)

functions using CCD technology. The architectures are dynamically prgrammable at the user level by controlling the threshold of the gates used.

In this thesis we will review some of the existing MVL-PLA architectures and optimization techniques and will explore the idea of proposing some new PLA architectures using existing set of Current-Mode CMOS MVL operators.

1.1.4 Research Objectives

The objectives of the thesis are :

1. Analyse the existing techniques and explore the idea of developing some new techniques for MVL function synthesis using decomposition based mapping technique.

2. Analyse the existing architectures and explore the idea of developing some new MVL-PLA structures for the realization of MVL functions.

3. Analyse the existing set of MVL current mode CMOS circuit elements and operators and simulate them to validate their functionality.

Chapter 2

Background and Definitions

In this chapter, definitions of some important terms used in this thesis and background material for MVL are given.

2.1 Definitions

Let $X = \{x_1, x_2, ..., x_n\}$ be a set of n variables, where x_i takes on values from $R = \{0, 1, 2, 3, ..., r-1\}$. A function $f(X)$ is a mapping $f : R^n \Longrightarrow R$. The function $f(X)$ is called an n-variable r-valued function [31]. An r-valued system has r possible input states and, accordingly r^r single variable r-valued functions. Therefore, for r-valued and n-variable system, there are $r^{(r^n)}$ different possible functions. For example, for 2-variable 4-valued logic, there will be $4^{(4^2)}$ i.e. 4,294,967,296 possible functions.

The definitions of some MVL operations are given below

Definition 2.1

A *complement* of a logical value is defined as [1, 8, 31]

$$\overline{x} = (r-1) - x$$

i.e., the complement of a logic value x is obtained by substracting it from $(r-1)$. For example, for 4-valued logic, the complement of x=0 is 3 and the complement of x=1 is 2.

Definition 2.2

A *max* (maximum) is defined as [1, 8, 31]

$$max(x_1, x_2, ..., x_n) = x_1 \diamond x_2, ...\diamond x_n$$

$$= x_i \qquad \text{if } x_i \geq x_j \text{ for all } i \neq j \quad (i = 0, 1, ..., n-1)$$

It represents the largest x_i, where $x_i \in R$ and R=$\{0, 1, 2, ..., r-1\}$. The *max* is the generalization of the binary OR operation.

For example in four-valued logic, if the values of three variables, x_1, x_2 and x_3 are 0, 2, and 3 respectively then the maximum among these three variables will be max(0,2,3)=3.

Definition 2.3

A *min* (minimum) is defined as [1, 8, 31]

$$min(x_1, x_2, ..., x_n) = x_1 \bullet x_2, ... \bullet x_n$$

$$= x_i \qquad \text{if } x_i \leq x_j \text{ for all } i \neq j \quad (i = 0, 1, ..., n-1)$$

It represents the smallest x_i, where $x_i \in R$ and R=$\{0, 1, 2, ..., r-1\}$. The *min* is

the generalization of the binary AND operation.

For example in four-valued logic, if the values of three variables, x_1, x_2 and x_3 are 1,2, and 3 respectively then the minimum among these three variables will be $min(1,2,3)=1$.

Definition 2.4

A *tsum* (truncated sum) is defined as [1, 8, 31]

$$tsum(x_1, x_2, ..., x_n) = x_1 \uplus x_2, ... \uplus x_n$$
$$= min(x_1 + x_2 + ... + x_n, r-1)$$

where $x_1, x_2, ...x_n \in R$.

The *tsum* operation is performed by taking the minimum of summation of n values and $(r-1)$.

Considering $r = 4$, x_1=3, x_2=2, and $x_3 = 2$, then $tsum(3,2,2)=min(3+2+2, r-1) = min(7,3) = 3$.

Definition 2.5

A *modsum* (modulus sum) is defined as [1, 31]

$$modsum(x_1, x_2, ..., x_n) = x_1 \oplus x_2, ... \oplus x_n$$
$$= (x_1 + x_2 + ... + x_n) \ mod \ r$$

where $x_i \in R$.

It is obtained as the modulo r addition of n-variables. Considering 4-valued logic and if the values of x_1, x_2, and x_3 are 0,2, and 3 respectively then

$modsum(0,2,3)=(0+2+3) \bmod 4 = 1$.

Definition 2.6

A *window literal* of an MVL variable x is defined as [1, 31]

$$
\begin{aligned}
{}^{a}x^{b} &= (r-1) && \text{if } a \le x \le b \\
&= 0 && \text{if } x < a \text{ or } x > b
\end{aligned}
$$

where $a, b \in R$ and $a \le b$.

The symbol (0123) at the input positions shows the four possible values of input current. The symbol $\langle\rangle$ at the output represents the corresponding currents at these positions. Thus the symbol $\langle 3210 \rangle$ at the output means that there will be an output current of 3 when the input current is 0 and the output current of 2 when the input current is 1, etc.

Considering r=4, then for input x=(0123), ${}^{2}x^{3} = \langle 0033 \rangle$, ${}^{0}x^{2} = \langle 3330 \rangle$, ${}^{1}x^{1} = \langle 0300 \rangle$ and ${}^{1}x^{2} = \langle 0330 \rangle$.

Definition 2.7

A *literal* of MVL variable x is defined as [31]

$$
\begin{aligned}
k[{}^{a}\{x\}^{b}] &= k && \text{if } {}^{a}\{x\}^{b} = \textit{binary high} \\
&= 0 && \text{if } {}^{a}\{x\}^{b} = \textit{binary low}
\end{aligned}
$$

where

$$
\begin{aligned}
{}^{a}\{x\}^{b} &= \textit{binary high} && \text{if } a \le x \le b \\
&= \textit{binary low} && \text{otherwise}
\end{aligned}
$$

$a, b \in R$, $a \le b$ and the value of the *literal* $k \in \{1, 2, ..., r-1\}$.

Similarly a *complement of literal* is defined as

$$k\overline{[{}^{a}\{x\}^{b}]} = k \qquad \text{if } {}^{a}\{x\}^{b} = \textit{binary low}$$

$$= 0 \qquad \text{if } {}^{a}\{x\}^{b} = \textit{binary high}$$

For example for 4-valued logic, for input $x = (0123)$, $3[{}^{0}\{x\}^{1}] = \langle 3300 \rangle$, $2[{}^{2}\{x\}^{2}] = \langle 0020 \rangle$, $3\overline{[{}^{1}\{x\}^{2}]} = \langle 3003 \rangle$, and $1\overline{[{}^{0}\{x\}^{1}]} = \langle 0011 \rangle$.

Definition 2.8

A *universal literal* is a one variable r-valued input two-valued output function which is defined as [26].

$$X_i^{S_j} = r - 1 \qquad \text{if } X_i \in S_j$$

$$= 0 \qquad \text{if } X_i \notin S_j$$

For example, for 4-valued logic the *universal literal* $X_1^{\{1,3\}}$ takes a value 3 if $X_1 \in \{1,3\}$ and it is 0 if $X_1 \in \{0,2\}$.

Definition 2.9

A *clockwise cycle* operator is defined as [1, 8, 31]

$$x^{\rightarrow b} = (x + b) \; mod \; r$$

A *counter clockwise cycle* operator is defined as

$$x^{\leftarrow b} = (x - b) \; mod \; r$$

The *counter clockwise cycle* operator can be represented in *clockwise* form as

$$x^{\leftarrow b} = x^{\rightarrow (r-b)} = (x - b + r) \; mod \; r$$

where $b \in R$.

Let r=5 and input x=(01234), then $x^{\rightarrow 0} = \langle 01234 \rangle$, $x^{\rightarrow 1} = \langle 12340 \rangle$, $x^{\rightarrow 2} = \langle 23401 \rangle$ $x^{\rightarrow 3} = \langle 40123 \rangle$, $x^{\rightarrow 2} = \langle 34012 \rangle$, $x^{\rightarrow 4} = \langle 12340 \rangle$.

Definition 2.10

The *k-inverter* is defined as:

$$x^k = \begin{cases} k & \text{if } x = 0 \\ 0 & \text{if } x \neq 0 \end{cases}$$

Definition 2.11

A *product term* $P = \phi(x_1, x_2, \cdots, x_n)$ is defined as a *min* of a set of *literals* or a set of *cycle* operators [4]. For example, $(2[^1\{x_1\}^2]) \bullet (2[^0\{x_2\}^3])$ and $(x_1^{\rightarrow 2} \bullet x_2^{\rightarrow 3})$ represent product terms for 4-valued 2-variable functions. The map representation of these product terms are shown in Figure 2.1(a) and 2.1(b) respectively.

Definition 2.12

An *implicant* of a multiple-valued function $f(x_1, x_2, \cdots, x_n)$ is a product term $I(X)$ such that $f(X) \geq I(X)$ for each assignment of values x to variables in X [4]. For example in Figure 2.1(a), $(1[^1\{x_1\}^2]) \bullet (1[^1\{x_2\}^1])$ and $(1[^1\{x_1\}^1]) \bullet (1[^2\{x_2\}^2])$ represents some of the implicants.

Definition 2.13

An implicant I(X) of a function $f(x_1, x_2, \cdots, x_n)$ is said to be a *prime implicant* if

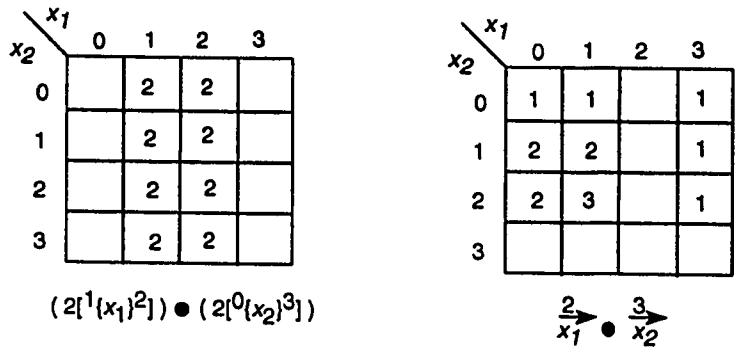


Figure 2.1: Map representation for some example product terms for 4-valued 2-variable functions.

there is no other implicant $I'(X)$ of $f(x_1, x_2, \cdots, x_n)$ such that $I'(x) \geq I(x)$ for all assignments of $x_i s$ [4]. For example in Figure 2.1(a) $(2[^1\{x_1\}^2]) \bullet (2[^1\{x_2\}^1])$ represents a prime implicant.

Definition 2.14

For MVL function, a function value $f(x)$ corresponding to a specific assignment of values x to variable in X is called a *minterm* such that $f(a_1, a_2, \cdots, a_n) \neq 0$, where, $a_1, a_2, \cdots, a_n$ represent the specific assignment of values to variables $x_1, x_2, \cdots, x_n$ respectively [4]. For an n-variable r-valued function there can be a maximum of r^n number of minterms.

Definition 2.15

A sum-of-products expression for $f(x_1, x_2, \cdots, x_n)$ is *minimal* if there is no other expression for $f(x_1, x_2, \cdots, x_n)$ which have fewer implicants [32]. For example, in Figure 2.1, $(2[^1\{x_1\}^2]) \bullet (2[^0\{x_2\}^3])$ represents the minimal expression.

Definition 2.16

For one-variable r-valued logic, a function $f(X)$ is an *increasing* function if and only if $a_0 \leq a_1 \leq \cdots a_{r-1}$, where $a_0 \leq a_1 \leq \cdots a_{r-1}$ represent the values of the function $f(X)$ at location $0, 1, \cdots, r-1$. Similarly a function is termed as *decreasing* if and only if $a_0 \geq a_1 \geq \cdots a_{r-1}$.

For example, for 4-valued one variable system, the function (0,1,1,3) represents an *increasing* function, whereas the function (3,2,1,0) is a *decreasing* function. The number of increasing function $N_{increasing}$ is give as [33]:

$$N_{increasing} = C \begin{pmatrix} 2r-1 \\ r-1 \end{pmatrix}$$

where C represent the combination of $(2r-1)$ things taking $(r-1)$ at a time. For example, for 4-valued one-variable logic, there will be 35 *increasing* functions. It is also known that the number of *decreasing* function is equal to the number of *increasing* function [33], and we have

$$N_{decreasing} = N_{increasing}$$

Definition 2.17

For one-variable r-valued logic, a function $f(X)$ is a *mixed* function if it is neither an *increasing* nor a *decreasing* function. For example, for one-variable 5-valued logic, the function (0,2,1,4,1) is a *mixed* function.

Definition 2.18

Given a two-variable r-valued function, represented as a map of order $r \times r$. A function is an *increasing* function if and only if it satisfies the following conditions:

(i) $a_{ij} \leq a_{i+1\,j+1} \leq a_{rr}$ where $i = j$

(ii) $a_{ij} \leq a_{i\,j+1} \leq a_{ir}$

(iii) $a_{ij} \leq a_{i+1\,j} \leq a_{rj}$

where $1 \leq i, j \leq r$. For example, the functions shown in Figure 2.2(a) and Figure 2.2(b) are 2-variable 3-valued and 4-valued *increasing* functions, respectively.

Similarly a function is *decreasing* if it satisfies the following conditions:

(i) $a_{ij} \geq a_{i+1\,j+1} \geq a_{rr}$ where $i = j$

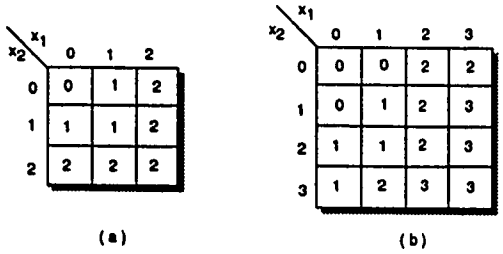
(ii) $a_{ij} \geq a_{i\,j+1} \geq a_{ir}$

(iii) $a_{ij} \geq a_{i+1\,j} \geq a_{rj}$

Figure 2.3(a) and Figure 2.3(b) show examples of 2-variable 3-valued and 4-valued *decreasing* functions, respectively.

Definition 2.19

A two-variable function is said to be *symmetric* if $f(x_1, x_2) = f(x_2, x_1)$ i.e., if all the elements along the diagonal of the map are same. As an example, Figure 2.4 shows a 2-variable 3-valued *symmetric* function. As it can be seen from the Figure 2.4, $f(1,0) = f(0,1) = 2$, $f(0,2) = f(2,0) = 1$ and $f(1,2) = f(2,1) = 1$.



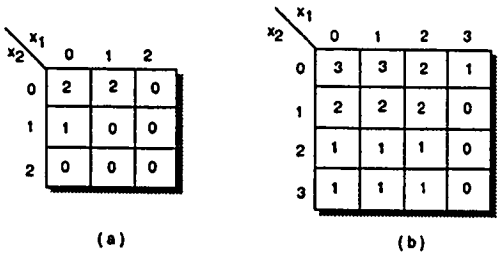
x_2 \ x_1	0	1	2
0	0	1	2
1	1	1	2
2	2	2	2

(a)

x_2 \ x_1	0	1	2	3
0	0	0	2	2
1	0	1	2	3
2	1	1	2	3
3	1	2	3	3

(b)

Figure 2.2: Example of *increasing* function for 2-variable: (a) 3-valued logic, (b) 4-valued logic.



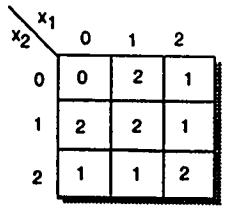
x_2 \ x_1	0	1	2
0	2	2	0
1	1	0	0
2	0	0	0

(a)

x_2 \ x_1	0	1	2	3
0	3	3	2	1
1	2	2	2	0
2	1	1	1	0
3	1	1	1	0

(b)

Figure 2.3: Example of *decreasing* function for 2-variable: (a) 3-valued logic, (b) 4-valued logic.



x_2 \ x_1	0	1	2
0	0	2	1
1	2	2	1
2	1	1	2

Figure 2.4: Example of *symmetric* function for 2-variable 3-valued logic.

2.2 Minimization heuristics for MVL functions.

Several algorithms have been proposed in the past for the synthesis of MVL functions. Generally, two approaches were used, namely *cost table* and *direct cover* approach. The cost table approach was first proposed by Kerkhoff and Robroek [34] for the synthesis of unary 4-valued functions. The cost table approach has been used to synthesize MVL functions at low cost. In this technique, a target MVL function is realized as the sum of a number of subfunctions each of which is in a table. The total cost of realization of a given MVL function is obtained by summing the costs of the sub-functions and the cost of combining them. The number of transistors required for realization is considered as the realization cost of the target function. The main disadvantage of the *cost table* approach is that it is difficult to obtain optimum cost tables for a given technology [4].

The *direct approach* proceeds in two steps. In the first step, a minterm is selected. Then an implicant that covers this minterm is selected. The implicant is then subtracted from the function. These steps are repeated until all the minterms of the given functions are covered [35].

In multiple-valued PLA, an MVL function is realized by combining one or more product terms using an operator such as *max* or *tsum*. Each of these product terms is realized as a single column in the PLA. Since all the columns have the same cost, minimizing the size of a multiple-valued PLA is equivalent to minimizing the

number of product terms required.

A brief review of some of the heuristics based on *direct cover* approach is given below:

Pomper & Armstrong Heuristic

Pomper and Armstrong [36] used the *min/tsum* based direct cover approach to obtain a near minimal realization. In their approach, they randomly selec a minterm and then selec an implicant which coveres the largest number of minterms i.e, the implicant that drives the largest number of minterms to 0 or don't care. If the number of such implicant is more than one, then the best implicant is chosen. A best implicant is one whose size is larger than the size of other implicants. Since an implicant can be written as $I = k[^{i_1}\{x_1\}^{j_1} \bullet^{i_2} \{x_2\}^{j_2} \ldots^{i_n} \{x_n\}^{j_n}]$, so by the size of an implicant it is meant the number of locations where this implicant assumes the value k. If there are more than one largest (best) implicant, then the implicant which is generated first is chosen. The process is stopped when all the functions are completely covered.

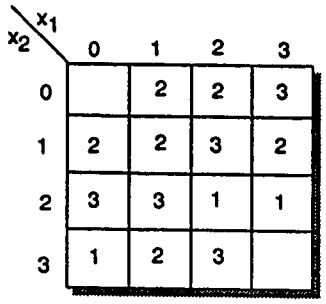
Dueck & Miller Heuristic

In Dueck and Miller approach, an isolated minterm is selected rather than a random minterm [37]. In this technique, the isolation factor is calculated for each minterm. The isolation factor gives a measure of the degree to which a minterm can combine

with other minterms. The minterm which has the highest isolation factor is selected. Then all the implicants are generated which cover this minterm. An implicant is selected which when subtracted from the function, introduces few discontinuities in the function. This makes the remaining function easy to realize.

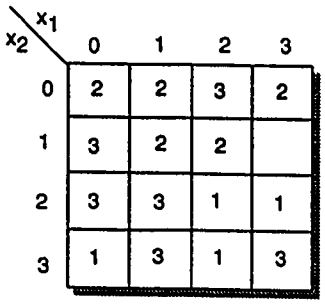
Generally, the Pomper and Armstrong, and Dueck and Miller algorithms perform better for *min/tsum* based realization. Yurchak and Butler [38] have developed a CAD tool called HAMLET (Heuristic Analyzer for Multiple-valued Logic Expression Translation) for *min/tsum* based realization. It is actually a family of software tools to design MVL-PLAs. It includes implementation for both Pomper and Armstrong (PA) and Dueck and Miller (DM) heuristics. The expression accepted by HAMLET is in the sum-of-products form, and the minimal expression can be obtained by applying the selected minimization heuristic.

It has been observed that no single algorithm is consistently better than the other over all functions. For example, Figure 2.5 shows a function where Dueck and Miller heuristic performs better than the Pomper and Armstrong heuristic. It requires 8 implicants to cover the functions, whereas Pomper and Armstrong heuristic requires 10 implicants. Similarly, Figure 2.6 shows the case where Pomper and Armstrong is superior than Dueck and Miller heuristic. It produces a realization that has two fewer implicants than that obtained by Dueck and Miller heuristic [7].



Pomper & Armstrong		Dueck & Miller	
Minterm	Implicant	Minterm	Implicant
$1[^2\{x_1\}^2]\bullet[^2\{x_2\}^2]$	$1[^1\{x_1\}^3]\bullet[^0\{x_2\}^2]$	$1[^0\{x_1\}^0]\bullet[^3\{x_2\}^3]$	$1[^0\{x_1\}^2]\bullet[^2\{x_2\}^3]$
$1[^0\{x_1\}^0]\bullet[^3\{x_2\}^3]$	$1[^0\{x_1\}^1]\bullet[^1\{x_2\}^3]$	$1[^3\{x_1\}^3]\bullet[^2\{x_2\}^2]$	$1[^3\{x_1\}^3]\bullet[^0\{x_2\}^2]$
$1[^1\{x_1\}^1]\bullet[^3\{x_2\}^3]$	$1[^1\{x_1\}^1]\bullet[^2\{x_2\}^3]$	$1[^1\{x_1\}^1]\bullet[^3\{x_2\}^3]$	$1[^1\{x_1\}^1]\bullet[^3\{x_2\}^3]$
$2[^0\{x_1\}^0]\bullet[^2\{x_2\}^2]$	$2[^0\{x_1\}^1]\bullet[^2\{x_2\}^2]$	$1[^3\{x_1\}^3]\bullet[^1\{x_2\}^1]$	$1[^2\{x_1\}^3]\bullet[^1\{x_2\}^1]$
$1[^2\{x_1\}^2]\bullet[^0\{x_2\}^0]$	$1[^2\{x_1\}^3]\bullet[^0\{x_2\}^1]$	$2[^2\{x_1\}^2]\bullet[^3\{x_2\}^3]$	$2[^2\{x_1\}^2]\bullet[^3\{x_2\}^3]$
$3[^2\{x_1\}^2]\bullet[^3\{x_2\}^3]$	$3[^2\{x_1\}^2]\bullet[^3\{x_2\}^3]$	$2[^3\{x_1\}^3]\bullet[^0\{x_2\}^0]$	$2[^1\{x_1\}^3]\bullet[^0\{x_2\}^0]$
$1[^1\{x_1\}^1]\bullet[^0\{x_2\}^0]$	$1[^1\{x_1\}^1]\bullet[^0\{x_2\}^0]$	$2[^2\{x_1\}^2]\bullet[^1\{x_2\}^1]$	$2[^0\{x_1\}^2]\bullet[^1\{x_2\}^1]$
$1[^3\{x_1\}^3]\bullet[^0\{x_2\}^0]$	$1[^3\{x_1\}^3]\bullet[^0\{x_2\}^0]$	$2[^0\{x_1\}^0]\bullet[^2\{x_2\}^2]$	$2[^0\{x_1\}^1]\bullet[^2\{x_2\}^2]$
$1[^2\{x_1\}^2]\bullet[^1\{x_2\}^1]$	$1[^2\{x_1\}^2]\bullet[^1\{x_2\}^1]$		
$1[^0\{x_1\}^0]\bullet[^1\{x_2\}^1]$	$1[^0\{x_1\}^0]\bullet[^1\{x_2\}^2]$		
10 implicants used		8 implicants used	

Figure 2.5: An example where Dueck and Miller heuristic is superior.



Pomper & Armstrong		Dueck & Miller	
Minterm	Implicant	Minterm	Implicant
$2[^2\{x_1\}^2] \bullet [^1\{x_2\}^1]$	$2[^0\{x_1\}^2] \bullet [^0\{x_2\}^1]$	$1[^3\{x_1\}^3] \bullet [^2\{x_2\}^2]$	$1[^0\{x_1\}^3] \bullet [^2\{x_2\}^3]$
$3[^0\{x_1\}^0] \bullet [^2\{x_2\}^2]$	$3[^0\{x_1\}^0] \bullet [^1\{x_2\}^2]$	$2[^3\{x_1\}^3] \bullet [^3\{x_2\}^3]$	$2[^3\{x_1\}^3] \bullet [^3\{x_2\}^3]$
$1[^2\{x_1\}^2] \bullet [^2\{x_2\}^2]$	$1[^0\{x_1\}^3] \bullet [^2\{x_2\}^3]$	$2[^1\{x_1\}^1] \bullet [^3\{x_2\}^3]$	$2[^1\{x_1\}^1] \bullet [^0\{x_2\}^3]$
$1[^2\{x_1\}^2] \bullet [^0\{x_2\}^0]$	$2[^2\{x_1\}^3] \bullet [^0\{x_2\}^0]$	$2[^2\{x_1\}^2] \bullet [^1\{x_2\}^1]$	$2[^2\{x_1\}^2] \bullet [^1\{x_2\}^1]$
$2[^3\{x_1\}^3] \bullet [^3\{x_2\}^3]$	$2[^3\{x_1\}^3] \bullet [^3\{x_2\}^3]$	$2[^3\{x_1\}^3] \bullet [^0\{x_2\}^0]$	$2[^3\{x_1\}^3] \bullet [^0\{x_2\}^0]$
$2[^1\{x_1\}^1] \bullet [^2\{x_2\}^2]$	$2[^1\{x_1\}^1] \bullet [^2\{x_2\}^3]$	$2[^0\{x_1\}^0] \bullet [^0\{x_2\}^0]$	$2[^0\{x_1\}^0] \bullet [^0\{x_2\}^2]$
		$1[^0\{x_1\}^0] \bullet [^1\{x_2\}^1]$	$1[^0\{x_1\}^0] \bullet [^1\{x_2\}^1]$
		$3[^2\{x_1\}^2] \bullet [^0\{x_2\}^0]$	$3[^2\{x_1\}^2] \bullet [^0\{x_2\}^0]$
6 implicants used		8 implicants used	

Figure 2.6: An example where Pomper and Armstrong heuristic is superior.

Chapter 3

Basic Current-Mode Circuit Elements

Electrical signals are used to carry information in an integrated circuit (IC). These electrical signals are represented by voltage, current, or charge depending on the technology used for circuit realization [8]. For example, in charge-coupled devices (CCDs), the packets of charge are moved under the control of applid voltage potential to carry the information. The amount of charge, voltage signal value, or current signal values represent the logic levels in an IC. Among these technologies, current-mode CMOS logic (CMCL) have been shown to be suitable for implementing MVL functions [39, 40]. This is due to the fact that they provide a powerful logic functionality and can utilize the existing binary logic circuits. Also for MVL circuits, the current-mode circuits provide more tolerance as compared to voltage-mode circuits.

The analog properties of current (summation, replication) also make it suitable to implement the threshold functions for performing arithmetical functions [41].

A combination of CMCL and voltage-mode CMOS logic(VMCL) techniques are used in a variety CMOS MVL circuit designs [31]. These circuit designs are discussed in this chapter. The voltage mode circuits normally consists of inverter, NAND, NOR, and some CMOS gates. The voltage-mode signals are binary and are referred to as *binary high* and *binary low* in the discussion. A brief discussion of the current-mode CMOS logic design is presented in the following section.

3.1 Current-mode CMOS Logic

In current-mode logic, current is used for the representation of multi-valued variables. The logic levels are represented using discrete current values. The logic levels are represented as an integer multiple of some reference current (I_o). For example, logic zero is represented by no current, logic 1 by I_o and logic 2 by current value $2I_o$, etc. In this thesis the reference current (I_o) has been assigned a current value of 20 μA [6]. The polarity of the logic value is determined by the direction of current flow. The current flowing from outputs to the inputs of the circuits are termed as *positive* current. For example, as shown in Figure 3.1 if the current flowing into the node n_1 is termed as *positive* current, then the current flowing out of the node n_1 is termed as *negative* current. A *positive* current represents a logic 1 (20 μA), whereas

Table 3.1: Truth table for a 4-valued inverter.

Input	Output
0	3
1	2
2	1
3	0

a *negative* current represents a logic -1.

Due to the analog properties of current, arithmetical operations such as addition and subtraction of signals are easy to perform in current-mode circuits. Also the current flowing into the circuits obey Kirchoff's current rule of electrical networks.

Let us consider the operation of a multiple-valued logic inverter, shown in Figure 3.1 [4] . The truth table of 4-valued logic inverter is shown in Table 3.1. The circuit consists of a current source that can supply $(r-1)I_o$ (i.e,$3I_o$) current. The input and output circuits act as a current sinks. When the input is at logic 0 , the input circuit can not sink any current (i.e $I_{in} = 0$). As a result there is no current flowing between nodes n_1 and n_2. All of the source current ($3I_o$) flows through the nodes n_2 to n_3 (i.e $I_{out} = 3I_o$). Thus, the output is at logic 3. However, when the input is at logic 1, the source current splits into two components, I_o current flows from n_2 to n_1 and $2I_o$ current flows from n_2 to n_3. Therefore, the output is at logic 2, which is in accordance with the truth table. The other states of the inverter can also be verified similarly.

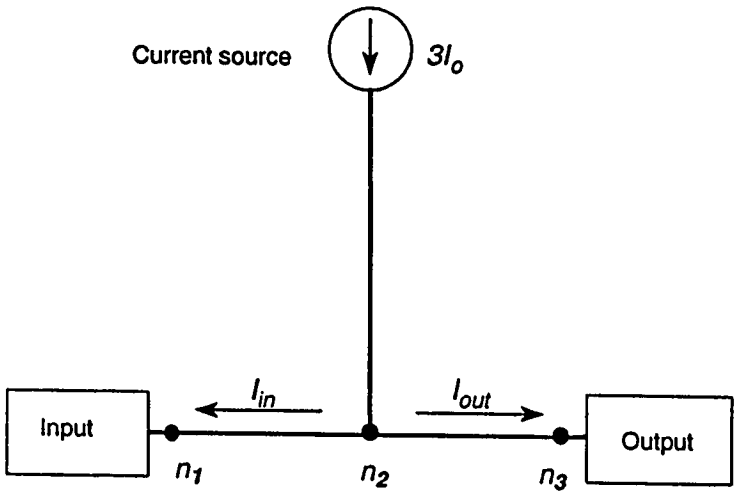


Figure 3.1: 4-valued CMCL inverter.

There are some disadvantages of current-mode CMOS circuits as compared to voltage-mode circuits. For example, the current-mode circuits have static power dissipation. Also the fanout is limited to one and the noise margin is poorer for higher radix current-mode circuits for a given range of current values. However, the above disadvantages can be overcomed by the fact that in current-mode circuits summation and subtraction operation can be performed easily. Furthermore, most current-mode CMOS MVL circuits have the advantage that they operate properly at reduced CMOS power supply voltages [6].

The basic circuit elements of current-mode CMOS are: *sum, constant, current-mirror, threshold,* and *switch.* Different CMOS MVL circuits have been designed by utilizing CMCL circuits along with VMCL circuits. In the following sections, the logic operation, symbol, and circuit realization for each of the above mentioned

elements will be discussed. The simulations of these circuit elements have been carried out to validate their functionality.

3.1.1 Sum

The logic operation of the *sum* circuit element is given by [4]

$$y = \sum_{i=1}^{n} x_i$$

where x_i represent the i_{th} input current and y represents the summation. The sum element consists of a number of input branches and an output branch. Figure 3.2 shows the circuit realization and symbol of the *sum* circuit.

Simulation results of *sum* circuit has been shown in Figure 3.3. Figure 3.3(a) and Figure 3.3(b) represent the two input currents whereas Figure 3.3(c) represents the summation of the two input currents.

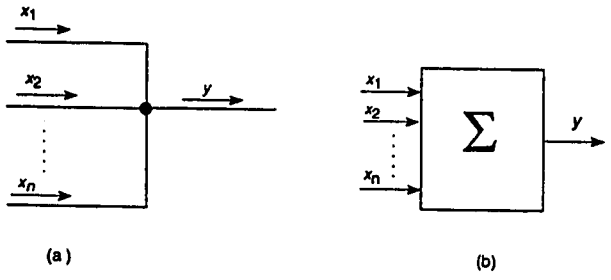


Figure 3.2: *sum* circuit element: (a) Circuit realization, (b) Symbol.

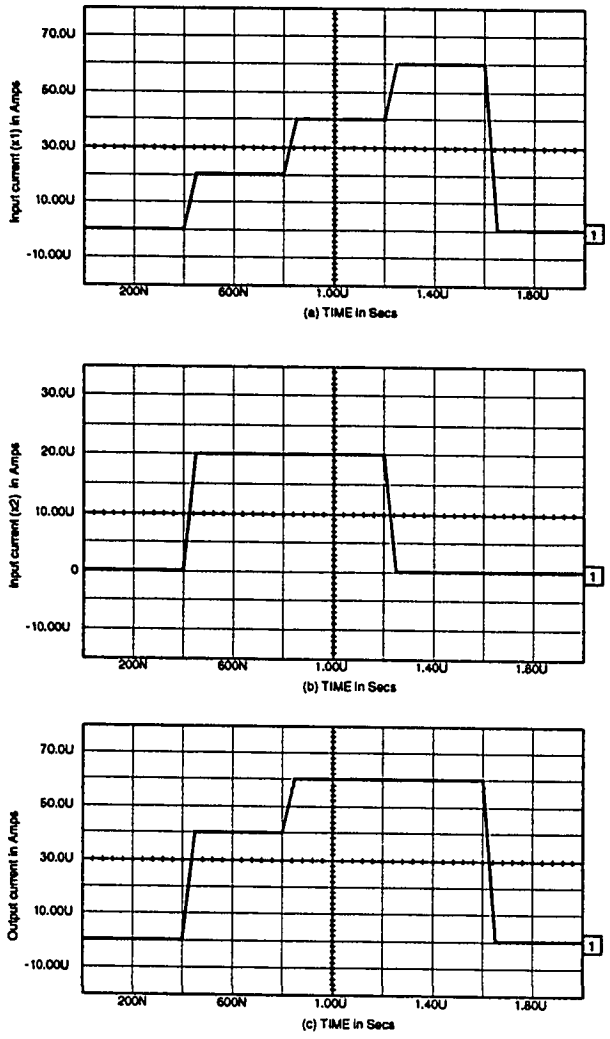


Figure 3.3: Transient analysis of $sum(x_1, x_2)$ as a function of time.

3.1.2 Constant

The *constant* circuit element is a current source which can be generated using a number of enhancement mode n-type or p-type transistors. It is used to generate logic values in MVL circuits by varying the width of the transistor. The logic operation for the constant circuit element is represented as [4]

$$y = k$$

where y is the output and k represents the width to length (W:L) ratio of the output transitor such that $k \in \{0, 1, 2, ..., r-1\}$ for r-valued logic. Figure 3.4 and Figure 3.5 show the circuit realization and symbol for n-type and p-type constant circuit element, respectively. In order to generate, $k = 1$ (i.e., $I_o = 20\mu A$) it is necessary to have N_{ref}=1.61V for an n-type transistor and P_{ref}=2.43V for a p-type transistor, and the size of each transistor should be (W:L=1:1).

The simulation for n-type constant element is shown in Figure 3.6. The size of the n-type transistor is set equal to $\frac{W}{L} = 2$. Since the reference current is $I_o = 20\mu A$, therefore, the constant circuit element gives an output current of $42.2\mu A$.

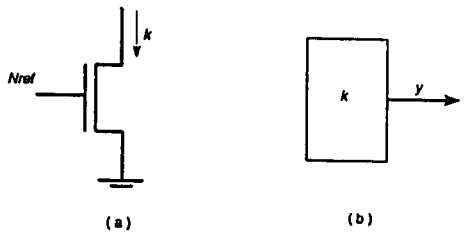


Figure 3.4: n-type constant circuit element. (a) Circuit realization. (b) Symbol.

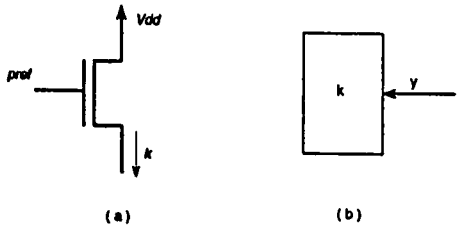


Figure 3.5: p-type constant circuit element.(a) Circuit realization. (b) Symbol.

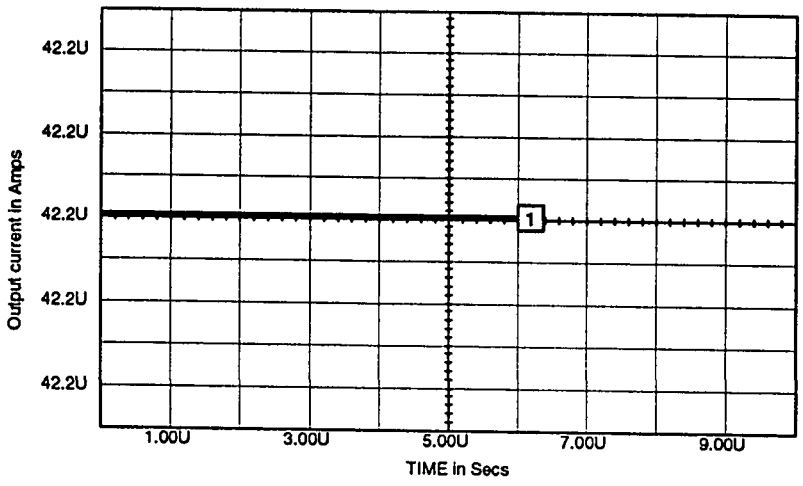


Figure 3.6: Simulation of n-type constant circuit element (W:L= 6:3).

3.1.3 Mirrors

Current mirrors are used as replicators of input current signals in MVL circuits. The current mirrors derive a set of output branches which is obtained by multiplication of the input current by some factor. The two types of current-mirrors, n-type and p-type are shown in Figure 3.7 and Figure 3.8, respectively. In these circuits, T_o

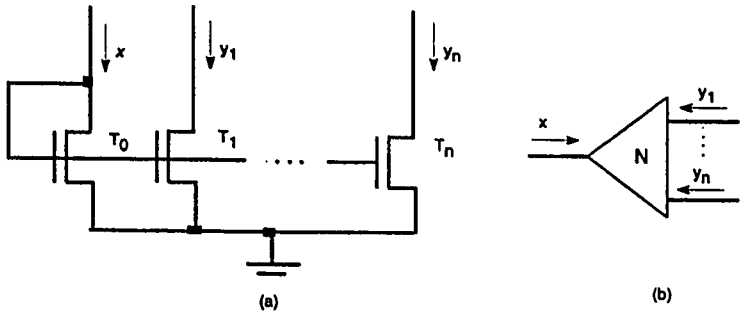


Figure 3.7: n-type *current mirror.* (a) Circuit realization. (b) Symbol.

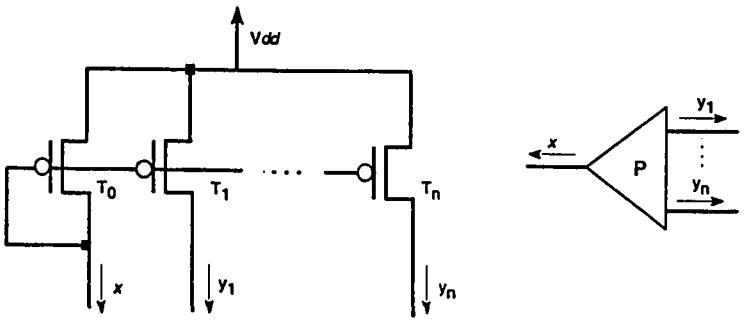


Figure 3.8: p-type *current mirror* circuit element. (a) Circuit realization. (b) Symbol.

represents the input transistor and $T_1, T_2, \cdots, T_n$ are the output transistors. The

logical operation of the current mirror is represented by the expression

$$y_i = a_i x$$

where y_i represents the output of ith transistor, x represents the input current and a_i is called the *scale factor* for the ith output. From the above equation it is necessary that the size of the ith output transistor should be $\frac{W_i}{L_i} = a_i \frac{W_o}{L_o}$. For example, if the size of the input transistor (T_o) is (W:L=3:3), then in order to get twice of the input current (I_o) at the output of transistor (T_2), the size of the output transistor should be (W:L=6:3).

Current mirrors are used to solve the problem of fanouts in CMCL circuits. It provides a fanout of more than one. Since, the direction of the current can be changed easily by current-mirrors, therefore, they are useful for addition and subtraction operation.

The simulation results for n-type mirror is shown in Figure 3.9. Figure 3.9(a) shows the input current where the size of the input transistor is (W:L=3:3). Figure 3.9(b) represents the output current where the size of the output transistor is twice that of the input transistor and as such the current is replicated by that amount. Similarly, Figure 3.9(c) represents the current of second output transistor of the current mirror which is three times the amount of input current.

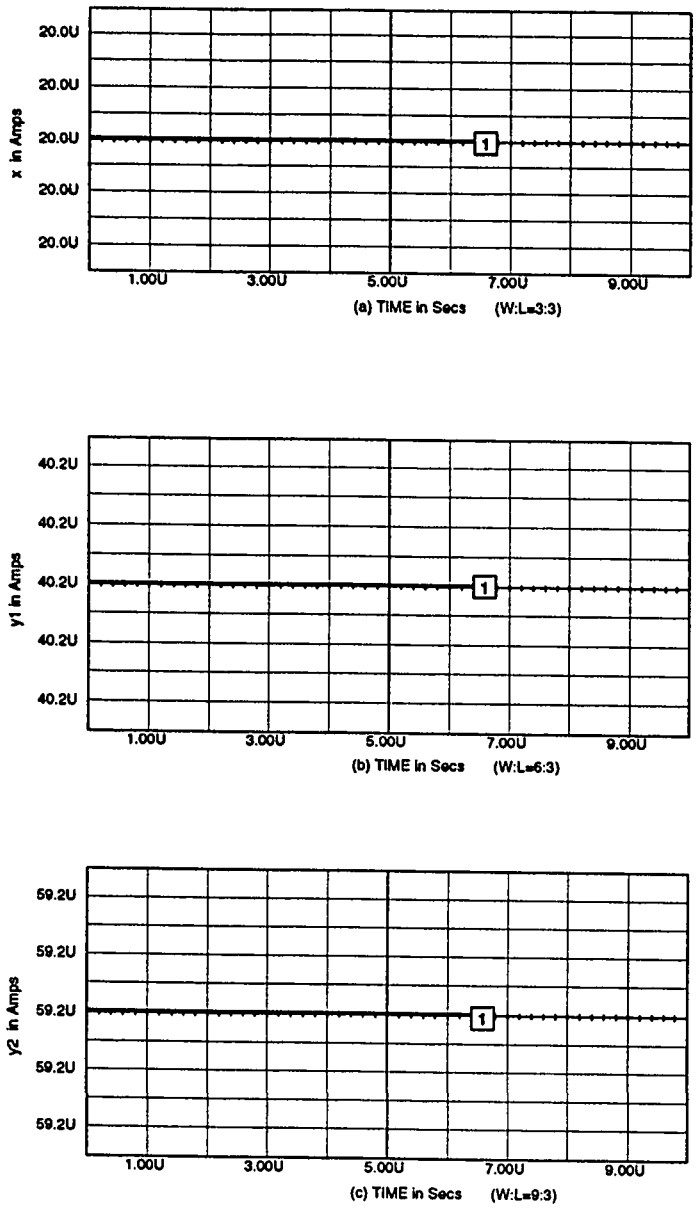


Figure 3.9: Simulation results of n-type mirror.

3.1.4 Threshold

The *Threshold* circuit element is used to convert current signal into binary voltage signal. The two types of threshold elements: n-type and p-type are shown in Figure 3.10 and Figure 3.11, respectively. The function of the n-type threshold element is defined as

$$\begin{aligned} y = thresh(x,k) &= \textit{binary high} && \text{if } x \geq k \\ &= \textit{binary low} && \text{if } x < k \end{aligned}$$

where x represents the input current and k represents the threshold value. When the input current, x is greater than some specified current level for n-type threshold value, k, then the output voltage signal, y, is *binary high* . The output voltage is *binary low* if the input current is less than the threshold value.

Similarly the logic function of the p-type threshold element is defined as

$$\begin{aligned} y = thresh(x,k) &= \textit{binary low} && \text{if } x \geq k \\ &= \textit{binary high} && \text{if } x < k \end{aligned}$$

The value of k depends on the size of the transistors and is equal to $(W/L)I_o$, where W is the width and L is the length of the transistor.

The simulation of n-type *threshold* element shown in Figure 3.10 are carried out to verify its functionality. The size of the transistor T_1 is set equal to (W:L = 3:2), i.e., $k = 1.5I_o$. For transient analysis, the input x was varied from logic 0 through to logic 3. These results are shown in Figure 3.12. Figure 3.12(a) represents variation

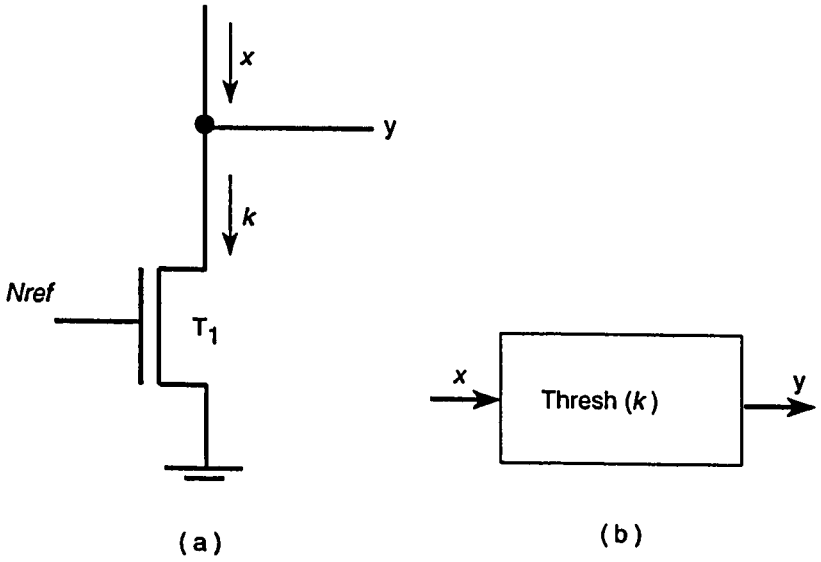


Figure 3.10: n-type *threshold* circuit element. (a) Circuit realization. (b) Symbol.

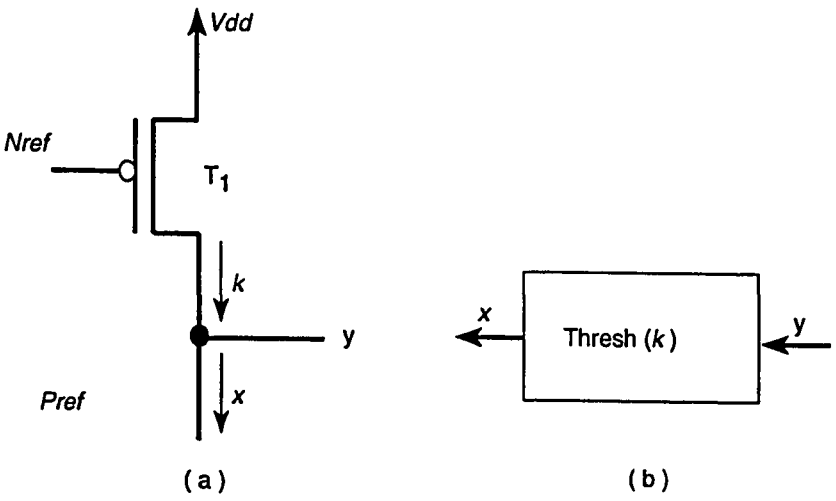


Figure 3.11: p-type *threshold* circuit element. (a) Circuit realization. (b) Symbol.

of the input current x and Figure 3.12(b) represents the voltage variation at node y, as a function of time. When the input x switches from logic 0 to logic 1, the voltage at node y remains at *Gnd*. The voltage at node y reaches *Vdd* only when input $x \geq 30\mu A$ (i.e.,$1.5I_o$). Therefore, when the input switches from logic 1 to logic 2 and logic 2 to logic 3 , the voltage at node y changes from *Gnd* to *Vdd*. This is in accordance to the definition of the *threshold* circuit element.

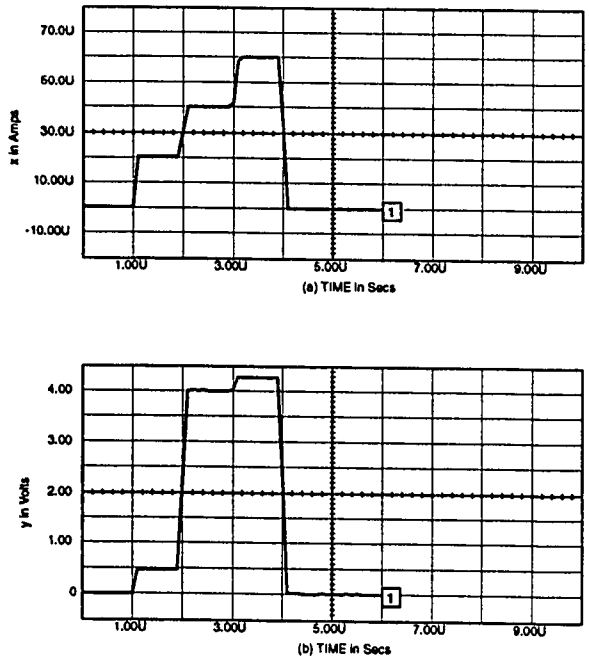


Figure 3.12: Transient analysis of n-type *threshold* circuit element. (a) Input current. (b) Output voltage.

3.1.5 Switch

switch allows the flow of current through an n-type or p-type transistor. This flow of current depends on the voltage signal at the gate input, V_{in}. The n-type and p-type switches are shown in Figure 3.13. The current flows through the transistor if the *switch* is ON, otherwise no current flows. The *switch* is generally used with the compatible type of circuit element. For example, if n-type *switch* is used with the n-type *constant* or *threshold* element, it allows a wider current range as compared to the one obtained using p-type *switch* . Similarly, the p-type circuit element performs better with p-type *switch.*

The simulation of the n-type *switch* circuit shown in Figure 3.13(a) has been carried out to verify its functionality. It can be observed from the simulation results shown in Figure 3.14 that when the voltage *Vdd* is applied to the *switch*, the output current follows the input current. Figure 3.15 show the simulation results when the applied voltage is zero. Similar results can be obtained for p-type *switch.*

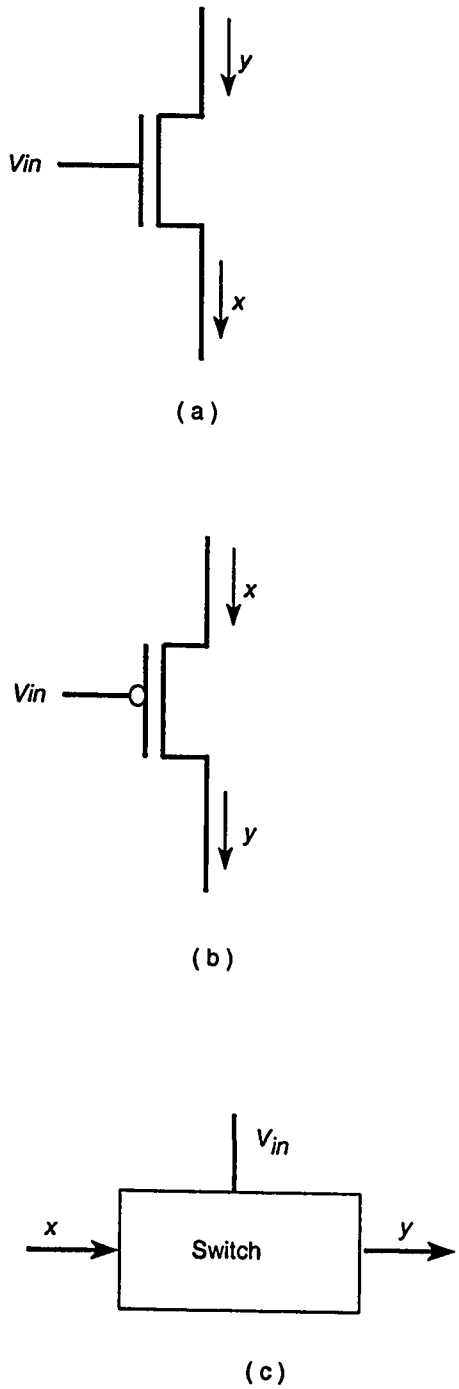


Figure 3.13: *switch* circuit element. (a) n-type circuit element. (b) p-type circuit element. (c) Symbol.

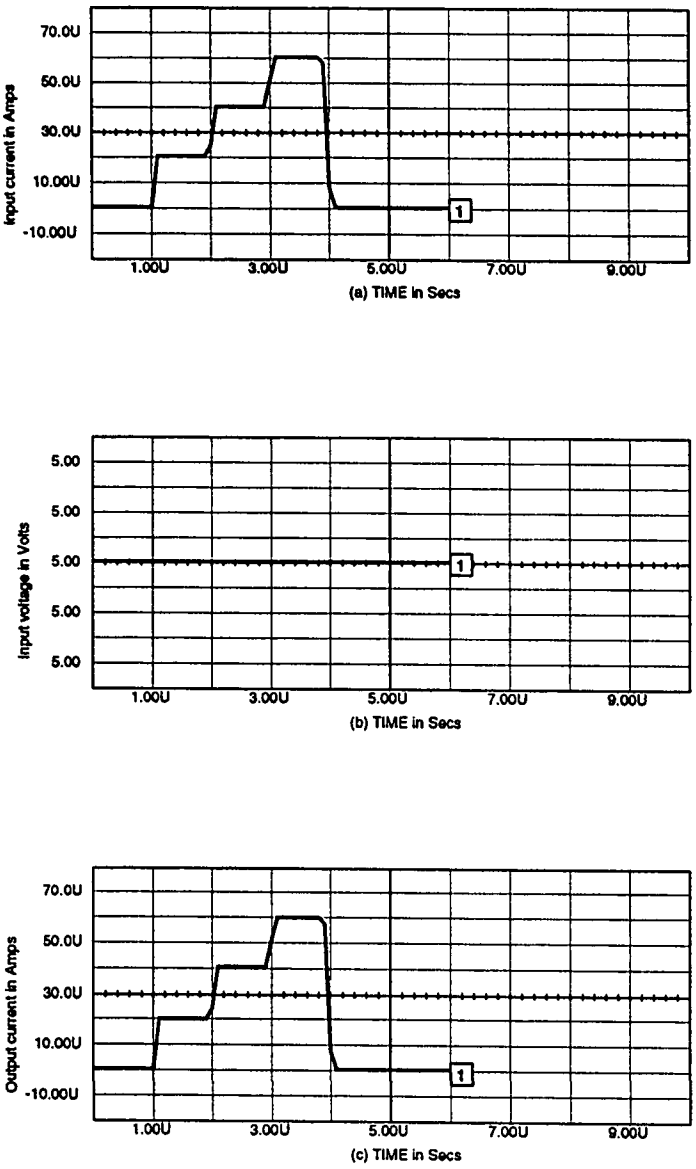


Figure 3.14: Simulation of n-type *switch* for $V_{in} = Vdd$.

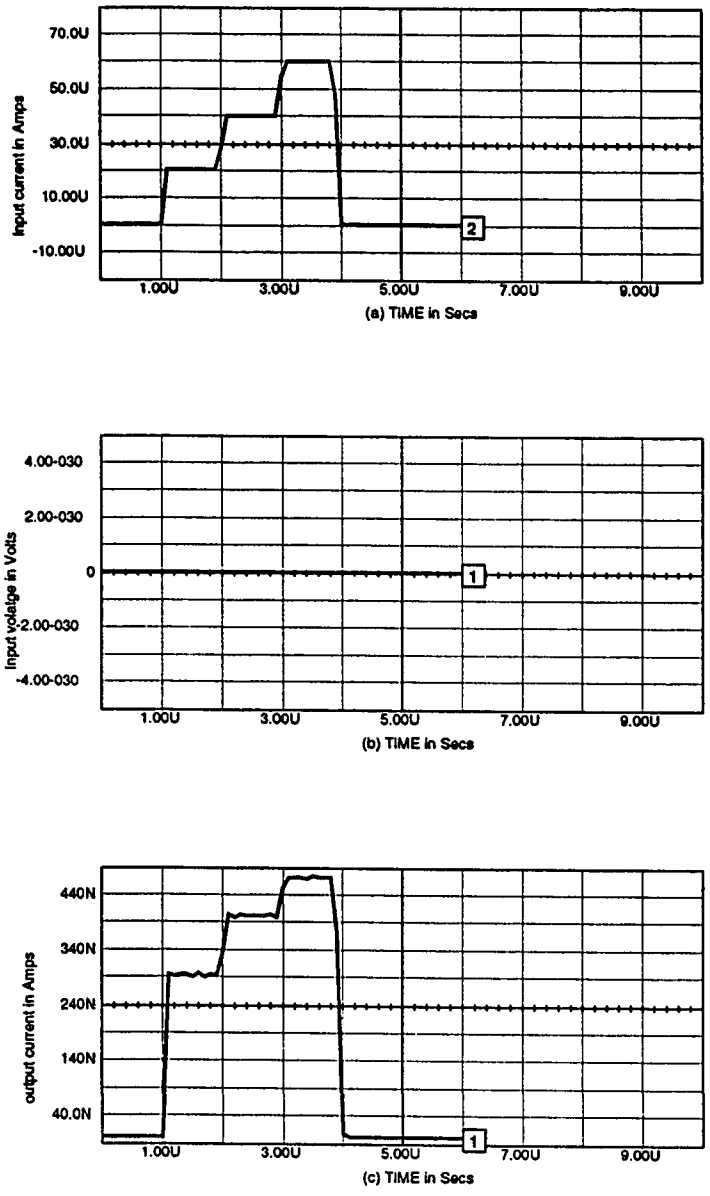


Figure 3.15: Simulation of n-type *switch* for $V_{in} = 0$.

3.2 Multiple-Valued Logic Operator Circuits

A set of operations is functionally complete if any given function can be represented by using only operators from this set. There are many ways of achieving functional completeness. Post [42] had proved that *cycle*, *min*, and *max* form a functionally complete set of operators. In 1968, Allen and Givone [43] had proposed an algebra consisting of *window literal*, *min*, and *max* operators. In 1970, Vranesic et al. [44] had proposed an algebra which was functionally complete using the operators *cycle*, *min*, *max*, and *unary inverter*. Furthermore, it has been shown that *window literal*, *min* and *tsum* are functionally complete set of operators [45].

In this chapter a functionally complete set of operators is presented for the realization of the MVL functions. These operators are selected from existing set of operators. These operators act as efficient building blocks for circuit realization in CMOS. The set includes: *cycle, tsum* and *min*. The circuit realizations use both the current-mode and voltage-mode CMOS circuit elements. MVL levels are represented by current-mode signals whereas the switches are controlled by the binary voltage-mode signals.

3.2.1 Circuit Implementation

In the following sections, the realization of some typical MVL circuits will be shown.

Min Operator

The *min* operator is realized based on the definition of the *truncated difference* operator as [46]:

$$x \ominus y = (x - y) \qquad \text{if } x \geq y$$

$$= 0 \text{ otherwise}$$

The *min* operator is defined as:

$$min(x, y) = x - (x \ominus y) = y - (y \ominus x)$$

The *min* operator has two realizations. Figure 3.16(a) represents the *min* circuit when the input currents are sinking and Figure 3.16(b) shows the circuit realization of *min* when the input currents are sourcing. These realizations are based on CMOS current-mode logic [31].

In these circuits all the n-type and p-type current mirrors have scale factor of unity. By applying the Kirchoff's current law at node A, we get $i_2 = i_3 = (x \ominus y)i_o$. Similarly by applying the Kirchoff's current law at node B we obtain $I_4 = xi_o - i_3$. The operation of the circuits shown in Figure 3.16 can be explained by considering the two cases [31]:

Case I: If $x < y$ then $i_1 = xi_o$ and according to the definition of *truncated difference* operator $i_2 = 0$. Since current i_2 is a input to the mirror, therefore i_3 is also zero and $i_4 = xi_o - i_3 = xi_o$. The output of the circuit, therefore is $min(x, y) = x$.

Case II: If $x \geq y$ then $i_1 = xi_o$ and $i_2 = xi_o - yi_o = (x - y)i_o$. Since i_2 is input to the

mirror and the scale factor of the output transistor is unity, therefore, $i_3 = (x-y)i_o$ and $i_4 = xi_o - i_3 = xi_o - (x-y)i_o = yi_o$, and it results in $min(x,y) = y$.

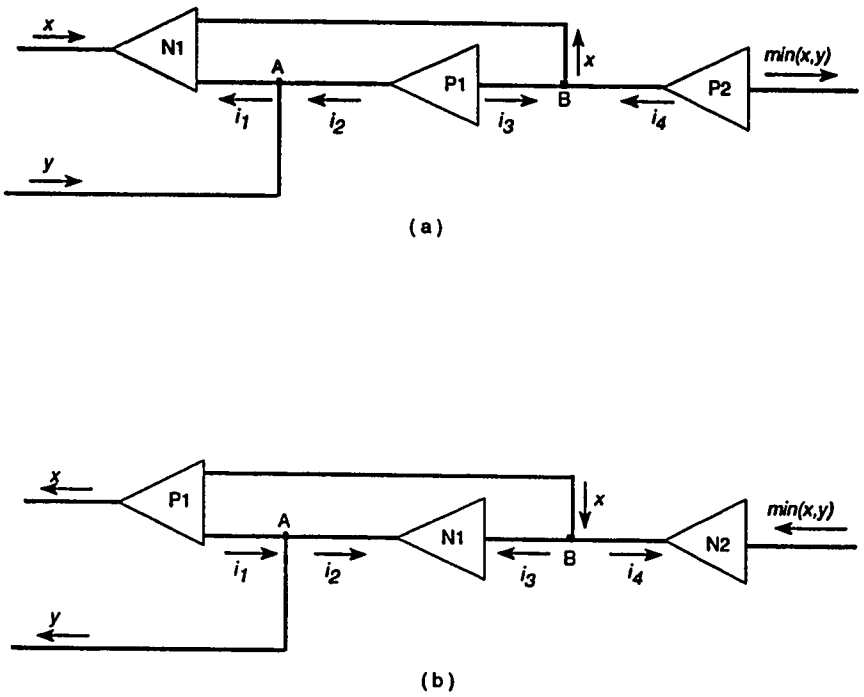


Figure 3.16: Circuit realizing *min* operator. (a) Input Currents are sinking. (b) Input currents are sourcing [4].

Tsum Operator

For r-valued n-variable function, the *tsum* operator is realized by the definition $min(x_1 + x_2 + \cdots + x_n, r-1)$, as shown in Figure 3.17 [31]. In this circuit, the input block consists of a p-type mirror and a n-type threshold element. The output block consist of two switches S_1 and S_2 and a p-type constant element. The control block is located between input and output block and it consists of an *inverter* . This *inverter* provides binary voltages to switches S_1 and S_2 of the output block. The summation of all currents X is applied as input to the p-current mirror which has a scale factor of unity. The transistor T_1 acts as a threshold element in the input block. In order to detect logic levels greater than $r-2$, the size of this transistor is set equal to $\frac{W_1}{L_1} = (r-1-0.5)$. The size of the transistor T_2 which acts as a p-type constant element, is set equal to $\frac{W_2}{L_2} = (r-1)$. This constant circuit element provides a current $i_2 = (r-1)i_o$ to the circuit when switch S_2 is ON. The operation of the circuit can be explained by considering the two cases i.e., when $0 < X < (r-1)$ and $(r-1) \leq X$.

Case I: $0 < X < (r-1)$ i.e., When the input current X is less than logic level $(r-1)$ then $i_1 = Xi_o$. Since i_1 is less than the specified current level at transistor T_1, therefore, the voltage at node A is binary low. Due to binary inverter, the voltage at node B is binary high and as a result *switch* S_1 is ON and the *switch* S_2 is OFF. Since *switch*, S_1 is ON and *switch*, S_2 is OFF, therefore $i_3 = Xi_o$ and $i_4 = 0$,

and the output current i_5 becomes $i_3 + i_4$, which is equal to Xi_o. The output logic level is X(i.e., $x_1 + x_2 + \cdots + x_n$).

Case II: $(r-1) \geq X$ i.e., When then input current X is more than the logic level $(r-1)$ then $i_1 = (r - 1 - 0.5)i_o$ and since T_1 is acting as a *threshold* element, therefore the voltage at node A is binary high and the voltage at node B is binary low. Due to the voltages at nodes A and B, the *switch* S_1 is OFF and the the *switch* S_2 is ON. Since S_1 is OFF, therefore the current $i_3 = 0$. The *switch* S_2 allows the current i_2 to pass through such that $i_4 = i_2 = (r-1)i_0$. The output current i_5 is equal to $i_3 + i_4$, which is $(r-1)i_0$. The output logic level becomes $(r-1)$ (i.e.,$min(x_1 + x_2 + \cdots + x_n, r-1)$).

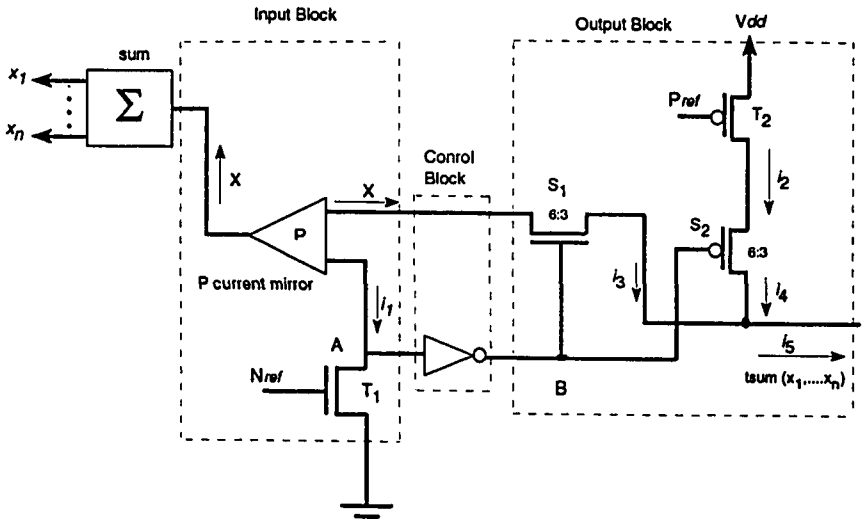


Figure 3.17: Circuit realization of *tsum* [31].

Cycle Operator

The two types of cycle operators: *clockwise* and *counter-clockwise* are related by [44]:

$$x^{\rightarrow^b} = x^{\leftarrow^{(r-b)}}$$

Because of the above equation, it is sufficient to consider only one type of *cyclic* operator. Figure 3.18 shows the circuit implementation of the *cycle* operator ($x^{\leftarrow^b}$). Transistors T_4, T_5, and T_6 act as p-type *current mirror*. The sizes of all three transistors of the *current mirror* are (W:L=3:3). The output of these current mirrors are the same as input current since the scale factor is unity. Transistor T_1 functions as a *threshold* element in the input block. The size of the transistor T_1 is set equal to $(b - 0.5)I_o$ to detect logic levels greater than b. The size of transistors T_2 and T_3 are $\frac{W_2}{L_2} = r - b$ and $\frac{W_3}{L_3} = b$, respectively. These transistors allow the current $(r - b)$ and b to pass through when switches S_2 and S_1 are ON. To understand the operation of the circuit consider the following two cases: $0 \leq x < b$ and $b \leq x < r$.

Case I: $0 \leq x < b$ i.e., When the input current x is less than b then $i_1 = xi_o$. Since this current is less than the threshold value $(b-0.5)$, therefore, the voltages at nodes A and B are binary low. The voltage at node B turns the *switch*, S_1 OFF. Similarly, since S_2 is a p-type *switch*, the binary low voltage at node A turns it ON. Under this condition i.e., when S_2 is ON and S_2 is OFF, current $i_2 = (r - b)i_o$ and $i_3 = 0$. If we apply the Kirchoff's current law at node C, the output current i_4 is equal to

$xi_o + i_2 - i_3 = xi_o + (r-b)i_o$. Thus, a logic level of $(r-b)$ is added to the input x to obtain the output, $x^{\leftarrow^b} = x - b + r$.

Case II: $b \leq x < r$ i.e., When then input current x is less than the logic value r or it is greater than or equal to b then the current i_1 becomes $(b-0.5)i_o$. Due to the threshold element T_1, voltages at nodes A and B are binary high. This makes the *switch*, S_1 ON and the *switch*, S_2 OFF. This allows the current $i_3 = bi_o$ to pass through the circuit whereas the current i_2 becomes zero. The output current i_4 is equal to $xi_o + i_2 - i_3$, which is less than the input xi_o by the value bi_o, and thus $x^{\leftarrow^b} = x - b$.

The size of transistors T_1, T_2, and T_3 can be varied to obtain circuit implementation for various values of b. Table 3.2 shows the size of all three transistors for different values of b for 4-valued logic.

Table 3.2: Size of the transistors T_1, T_2, and T_3 for the realization of *cycle* operator in 4-valued logic.

Cycle Operator	Size of Transistors $(\frac{W}{L})$ Ratios		
	$T_1(W_1 : L_1)$	$T_2(W_2 : L_2)$	$T_3(W_3 : L_3)$
$x^{\leftarrow^1}$	1:2	3:1	1:1
$x^{\leftarrow^2}$	3:2	2:1	2:1
$x^{\leftarrow^3}$	5:2	1:1	3:1

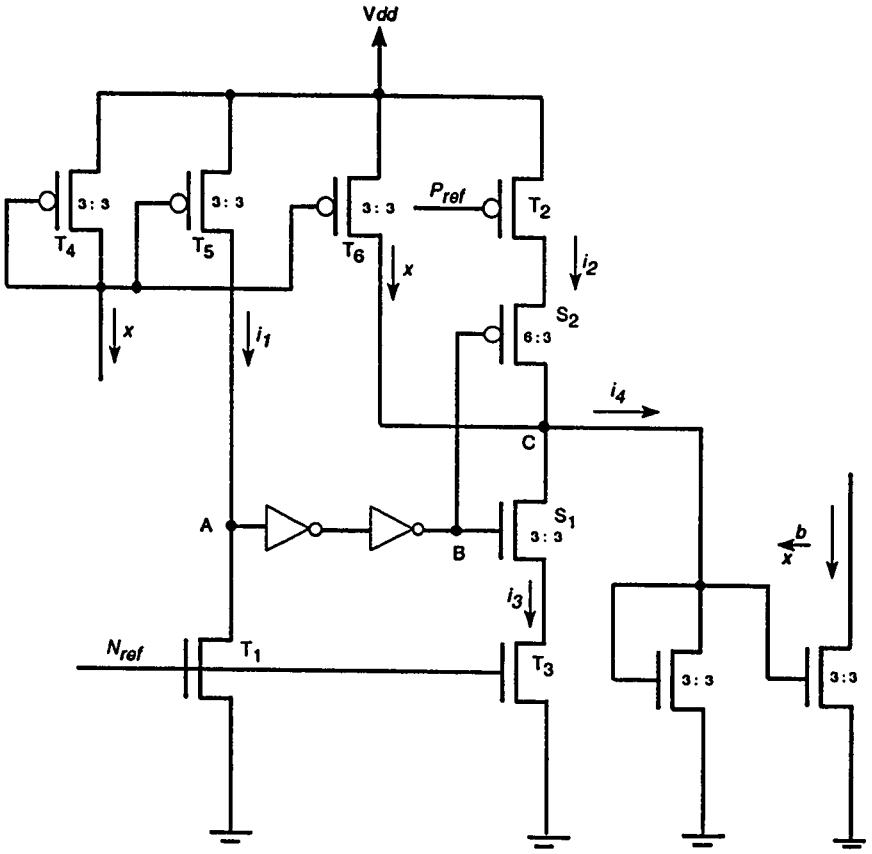


Figure 3.18: Circuit realization of *cycle* operator $x^{\leftarrow b}$ [31].

The *complement of cycle* operator is obtained by subtracting the *cyclic* operator by the amount $(r-1)$. The *complement of cycle* operator is obtained by the following relation:

$$\overline{x^{\rightarrow b}} = (r-1) - x^{\rightarrow b}$$

For example for 4-valued logic if x=(0123), then $x^{\rightarrow 2} = \langle 2301 \rangle$ and $\overline{x^{\rightarrow 2}} = \langle 1032 \rangle$. The relationship between *cycle* and *complement of cycle* operation is defined by the following equation:

$$\overline{x^{\rightarrow b}} = \overline{x}^{\rightarrow (r-b)} = \overline{x}^{\leftarrow b}$$

For example for 4-valued logic if input $x = (0123)$, then $\overline{x} = \langle 3210 \rangle$ and $\overline{x^{\rightarrow 2}} = \langle 1032 \rangle$ which is equal to $\overline{x}^{\rightarrow 2}$ and $\overline{x}^{\leftarrow 2}$. This implementation is useful if input variables and their complements are available.

3.3 Simulations of MVL circuits

To verify the functionality of the MVL circuits discussed in the previous section, SPICE transient analysis simulation was carried out. All these simulations were performed using SPICE MOS level 3 model. The Northern Telecom CMOS3DLM parameters were used in the simulation provided by Canadian Microelectronics Corporation (CMC).

A. Min Operator

Consider the realization of the *min* circuit shown in Figure 3.16 for 4-valued case. The simulation results are shown in Figure 3.19. The figure shows the variations of current level as a function of time. Figure 3.19(a) and Figure 3.19(b) represent the input currents and it show all possible combinations for the input variables, x_1 and x_2, respectively, while Figure 3.19(c) shows the corresponding output logic levels. The output logic levels are in accordance with the expected logic levels.

B. Tsum Operator

Consider the realization of the *tsum* circuit shown in Figure 3.17 for 4-valued case. The simulation results are shown in Figure 3.20. Figure 3.20(a) and Figure 3.20(b) represent the input currents as a function of time and it show all possible combinations for the input variables, x_1 and x_2, respectively. Figure 3.20(c) shows the output $tsum(x_1, x_2)$ as a function of time. As it can be seen from these figures that when input x_2 is at logic 0, the output follows the input x_1. When x_2 is at logic 3 (60 μA), the output is at logic 3 and it does not depend on the input current x_1. This shows that the circuit behaves in accordance to the definition of *tsum* operator. The simulations results can also be verified for other combinations of the input logic values.

C. Cycle Operator

Figure 3.21(a) and Figure 3.21(b) show the simulation results of $x^{\rightarrow^2} = \langle 2301 \rangle$ (i.e., $x^{\leftarrow^2}$) for 4-valued logic. In order to realize the circuit of *cycle* operator, $x^{\rightarrow^2}$, the size of the transistors T_1, T_2, and T_3 in Figure 3.18 should be set equal to $\frac{W_1}{L_1} = 1.5$, $\frac{W_2}{L_2} = 2$, and $\frac{W_3}{L_3} = 2$, respectively. The simulation results of $x^{\rightarrow^2}$ shown in Figure 3.21(a) and Figure 3.21(b) give the variations in current levels as a function of time. Figure 3.21(a) shows all the possible combination of logic levels for the input variable, x and Figure 3.21(b) shows corresponding variation for $x^{\rightarrow^2}$. As it can be seen from the Figure 3.21(a) and Figure 3.21(b) that for input at logic level 0 (0 μA) , 1 (20 μA), 2 (40 μA) and 3 (60 μA) the output is at logic level 2 (40 μA), 3 (60 μA), 0 (0μA) and 1 (20 μA), respectively. Thus from the simulation results, it is observed that the circuit perform in accordance to the definition of *cycle* operator.

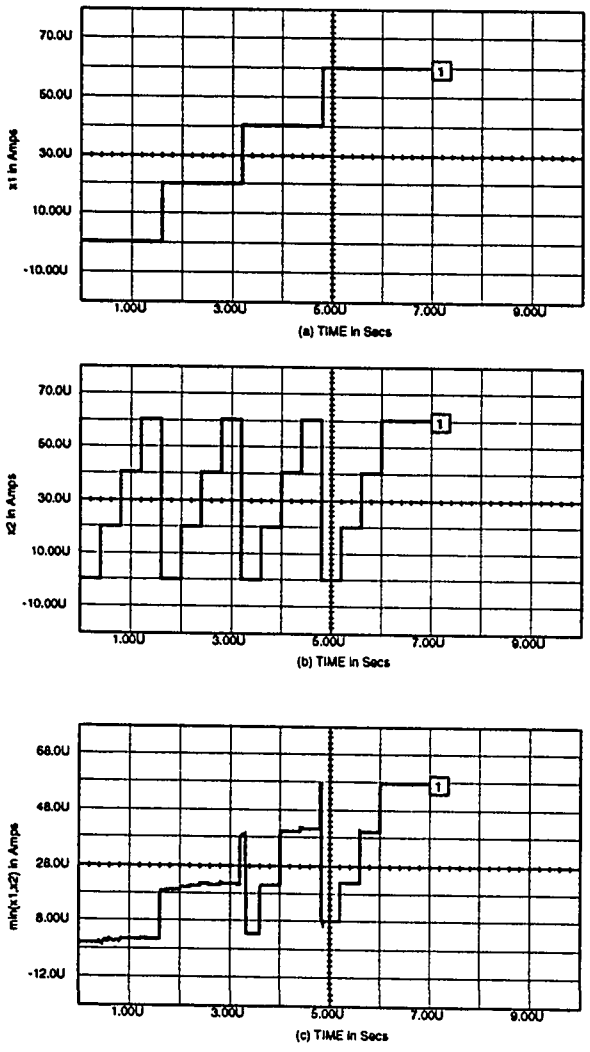


Figure 3.19: Transient analysis of the circuit realization of $min(x_1, x_2)$.

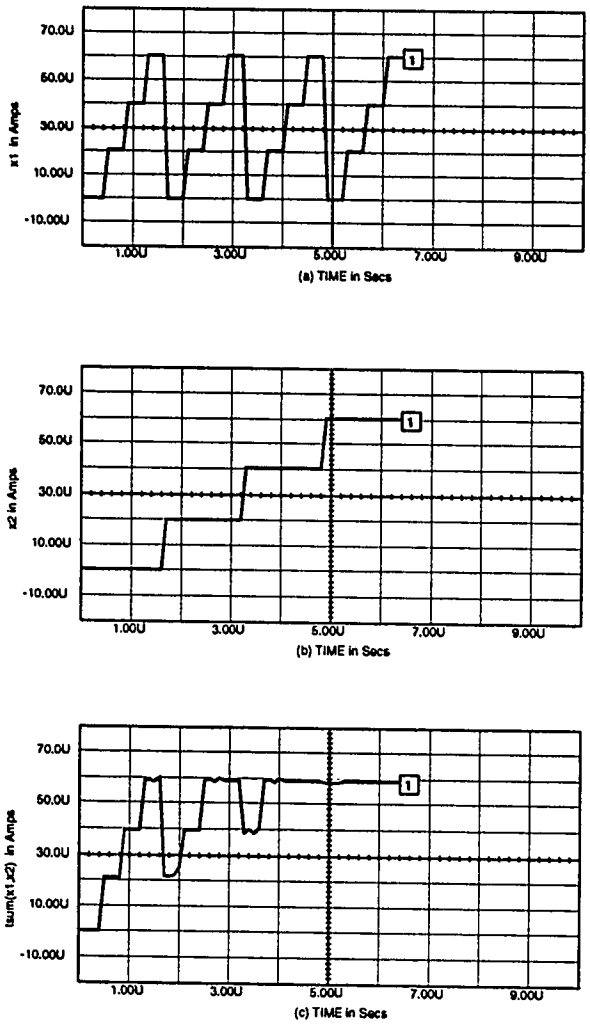


Figure 3.20: Transient analysis of the circuit realization of $tsum(x_1, x_2)$.

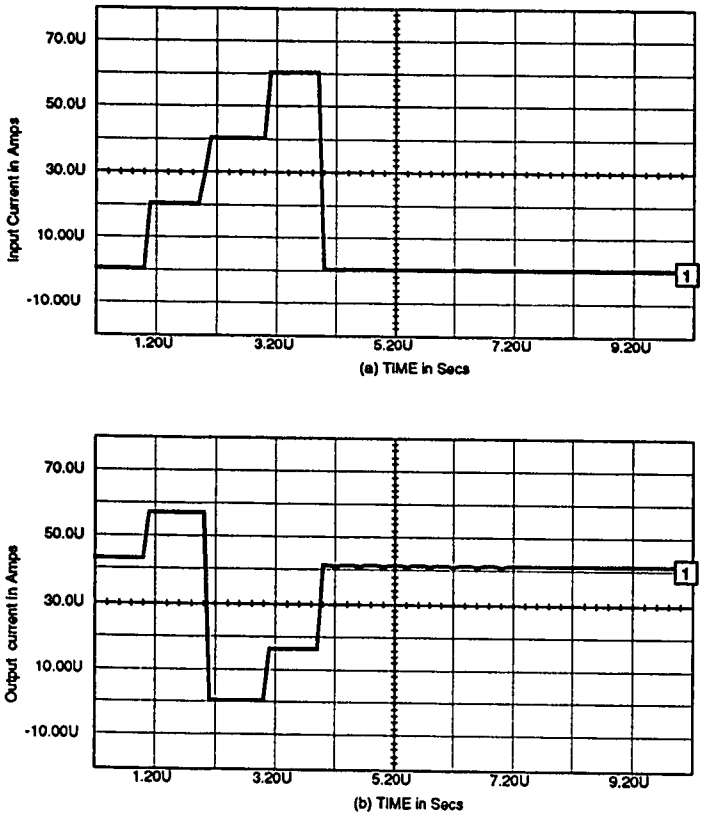


Figure 3.21: Transient analysis of the circuit realization of x^{-2}.

Chapter 4

Synthesis of MVL Functions

The design of logic networks has many constraints such as cost, speed, fan-in, fan-out, etc. The cost constraints led to the development of many simplification techniques. These simplification techniques are aimed at reducing the cost by minimizing the number of basic elements in the network.

Several synthesis techniques for the MVL functions have been presented in the past. In this chapter, a review of some of the existing synthesis techniques for MVL function will be presented.

4.1 A Functional Transformation Technique

Vranesic and Waliuzzaman [24] presented a functional transformation technique for the simplified implementation of the multi-valued functions. The functional

transformation is achieved by permuting the truth values. In this technique, the original function f was not implemented directly, rather another function g was obtained by permuting the truth values of the function f. This permuted function g provides a simpler implementation requiring smaller number of gates compared to the original function f. After implementing the permuted function g, the truth values of this function is permuted back to get the original function f. This technique did not provide much saving for binary implementation. However, for $R > 2$ this technique often provides reduction in cost. The implementation of the logic function is realized by using *min, max, cycle* and *k-inverter* gates. The definition of *min, max* and *cycle* has been given in Chapter 2.

Definition

The *precedence matrix* P of an r-valued function $f(X)$ is an $R \times R$ matrix whose element p_{ij} represents the number of times the truth value i is immediately followed by the truth value j when the function f is scanned in increasing order of truth values of its variables. The suffix i and j represents the ith row and jth column of the P matrix respectively, where $0 \leq i, j < R$. The P matrix is used to find the optimal permutation resulting in best functional transformation.

Definition

The *break count* of the P matrix is defined as the sum of all elements in the matrix

except those on the main diagonal or immediately above it. It is given by the relation

$$breakcount = \sum_{i=0}^{R-1} \sum_{j=0}^{R-1} p_{ij} \qquad j \neq i,\ j \neq i+1$$

It is known that functions which have lower breakcounts have simpler implementation than those functions which have higher breakcount value [24].

The functional transformation technique can be explained by considering the following example:

Example

Consider a 2-variable 4-valued function $f(x_1, x_2)$ whose truth table is shown in Figure 4.1(a). The implementation of the given function using the basic operators mentioned above can be achieved as follows:

$$f(x_1, x_2) = (x_1^{\leftarrow 3} \bullet x_2^{\leftarrow 3})^3 \diamond (x_1 \diamond x_2)^2 \diamond (x_1^{\leftarrow 2} \bullet x_2^{\leftarrow 2})^1$$

The corresponding realization is shown in Figure 4.2. It can be observed that 11 basic gates are required to implement the realization. The P matrix of the function $f(x_1, x_2)$ is shown in Figure 4.1(b). The breakcount of the P matrix is 8. Now the truth values of function $f(x_1, x_2)$ are permuted to get a new function $g(x_1, x_2)$. The permutations of the truth values are done as follows

$0 \Rightarrow 1$

$1 \Rightarrow 2$

$2 \Rightarrow 0$

$3 \Rightarrow 3$

The truth table of the permuted function $g(x_1, x_2)$ is shown in Figure 4.3(a). The permuted function $g(x_1, x_2)$ can be realized as follows

$$g = x_1 \diamond x_2$$

The P' matrix of the permuted function is obtained from the P matrix of the original function in such a way as to minimize the breakcount. If in the P matrix of original function $f(x_1, x_2)$, column(row) 0 and 1 and then column(row) 0 and 2 are interchanged, P' matrix of the permuted function $g(x_1, x_2)$ is obtained. This matrix has a break count of 0 as shown in Figure 4.3(b). Thus the permuted function $g(x)$ gives a simpler implementation. The original function $f(x)$ is obtained by reverse permuting the truth values of $g(x)$ as follows:

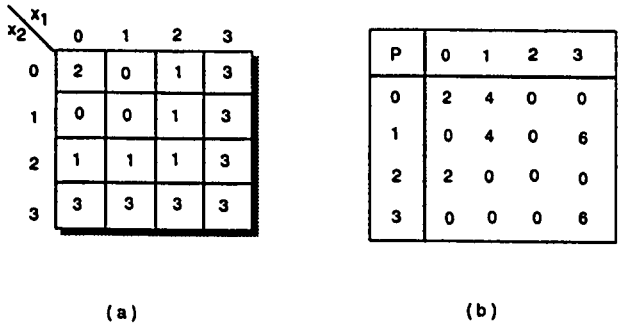
$$0 \Rightarrow 2$$

$$1 \Rightarrow 0$$

$$2 \Rightarrow 1$$

$$3 \Rightarrow 3$$

In general any mapping of truth values $M \Rightarrow N$ in the function g is obtained as $(g^{-M})^N$ terms. Thus after reverse permutation of $g(x)$, we have $f = g^2 \diamond (g^{-2})^1 \diamond (g^{-3})^3$. The implementation of this function is shown in Figure 4.4. It can be observed that only 7 gates are required for the realization as compared to 11 gates required for the original function $f(x)$, thus it reduces the cost significantly.



x_2 \ x_1	0	1	2	3
0	2	0	1	3
1	0	0	1	3
2	1	1	1	3
3	3	3	3	3

(a)

P	0	1	2	3
0	2	4	0	0
1	0	4	0	6
2	2	0	0	0
3	0	0	0	6

(b)

Figure 4.1: (a) Truth table for $f(x_1, x_2)$ (b) P matrix of $f(x_1, x_2)$.

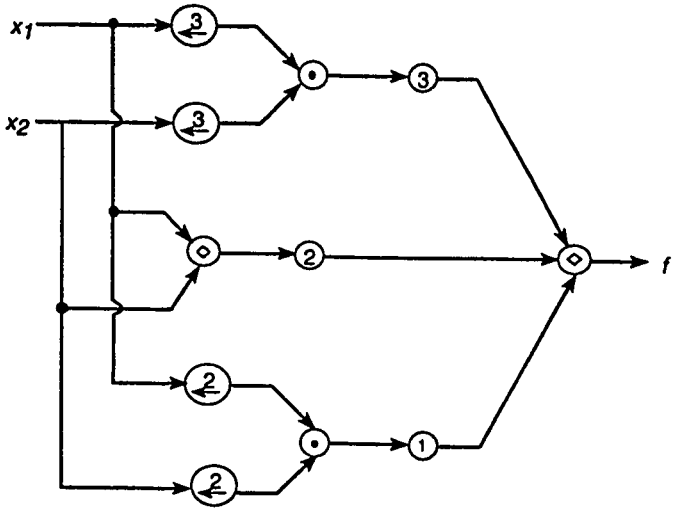
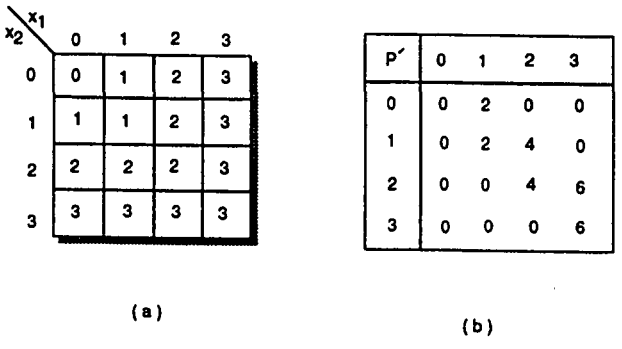


Figure 4.2: Direct implementation of $f(x_1, x_2)$.



x_2 \ x_1	0	1	2	3
0	0	1	2	3
1	1	1	2	3
2	2	2	2	3
3	3	3	3	3

(a)

P′	0	1	2	3
0	0	2	0	0
1	0	2	4	0
2	0	0	4	6
3	0	0	0	6

(b)

Figure 4.3: (a) Truth table for $g(x_1, x_2)$ (b) P matrix of $g(x_1, x_2)$.

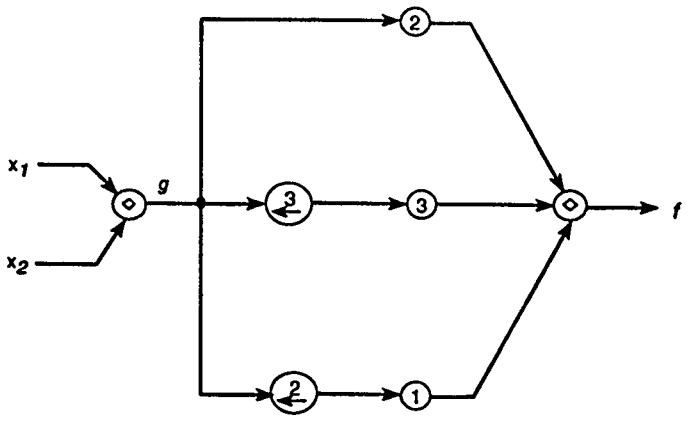


Figure 4.4: Transformed implementation of $f(x_1, x_2)$.

4.2 A Disjunctive-Decomposition Technique

Sami and Samuel [47] presented a fast algorithm for the disjunctive decomposition of r-valued functions. They generalized the Shen et.al [48] binary function decomposition algorithm to r-valued functions. It was based on the testing of necessary conditions for decomposability of MVL switching function. Partition tables have been used to test these conditions. It has been shown that for n-variable r-valued randomly selected functions, decomposability can be found in $(nr)^3$ time. Let us review some of the definitions used in [47]:

Definitions

1. An r-valued function $f(X)$ have a *simple disjunctive decomposition* if there exist r-valued functions g and h such that

$$f(X) = g(h(X_1), X_2)$$

where $\{X_1, X_2\}$ is a partition of X. X_1 and X_2 are called the *bound set* and *free set* respectively.

2. Given a partition $\{X_1, X_2\}$ of an r-valued function $f(X)$, a *partition table* of a function denoted by $T(X_1 : X_2)$ is defined as a truth table where X_1 and X_2 defines the columns and rows respectively.

3. The number of distinct columns in a partition table $T(X_1 : X_2)$ is called the column multiplicity and denoted by v.

4. The decomposition of a function is said to be *nontrivial* if the number of elements is not equal to 1 or n, where n is the number of variables.

5. The *partial partition table* $T(x_i, x_j : x_k, m)$ of a function $f(X)$ is defined as the partition table $T(x_1, x_j : x_k)$ of the function $f(x_i, x_j, x_k, m)$, where $0 \leq m \leq r^{n-3} - 1$.

A function that has a nontrivial simple disjunctive decomposition is said to be *decomposable.* A function has a nontrivial simple disjuntive decomposition only if the column multiplicity v of partition table $T(X_1 : X_2)$ is less than or equal to r. This condition is checked by generating all the partition table of the function under test and checking their column multiplicity. The other condition of decomposability of a fucntion is that all the partial partition tables must also have a column multiplicity $v \leq r$.

Example

Consider a 3-valued function $f(X)$ as shown in Figure 4.5(a). As it can be observed that the column multiplicity v is 3 which equals r, thus it satisfies the condition for decomposabilty. The partition table has a bound set $\{x_1, x_2, x_3\}$ and a free set $\{x_4\}$. The truth table of the function $h(x_1, x_2, x_3)$ and $g(h, x_4)$ has been shown in Figure 4.5(b) and Figure 4.5(c), respectively. Figure 4.6 shows the circut diagram of the decomposable function $f(X) = (h(x_1, x_2, x_3), x_4)$.

x_1	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2	2
x_2	0	0	0	1	1	1	2	2	2	0	0	0	1	1	1	2	2	2	0	0	0	1	1	1	2	2	2
x_4 \ x_3	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2
0	1	2	0	1	0	0	1	1	2	2	0	1	1	0	1	1	2	2	2	1	2	0	1	1	1	0	0
1	2	2	0	2	0	0	2	2	2	2	0	2	2	2	2	2	2	2	2	2	2	0	2	2	2	0	0
2	0	0	2	0	2	2	0	0	0	0	2	0	0	0	0	0	0	0	0	0	0	2	0	0	0	2	2

(a) $f(x_1, x_2, x_3, x_4)$

x_1	0	0	0	1	1	1	2	2	2
x_3 \ x_2	0	1	2	0	1	2	0	1	2
0	1	1	1	2	1	1	2	0	1
1	2	0	1	0	0	2	1	1	0
2	0	0	2	1	1	2	2	1	0

(b) $h(x_1, x_2, x_3)$

h \ x_4	0	1	2
0	1	2	0
1	2	2	0
2	0	0	2

(c) $g(h, x_4)$

Figure 4.5: Decomposition of $f(X) = g(h(x_1, x_2, x_3), x_4)$.

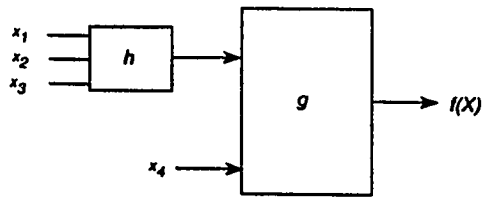


Figure 4.6: Circuit diagram of $f(X) = g(h(x_1, x_2, x_3), x_4)$.

4.3 Decomposition Based Mapping Technique

In [25] Abdel-Hamid and Abd-El-Barr have presented a decomposition-based mapping technique for the synthesis of binary and MVL functions. The synthesis problem is formulated as a mapping from an input matrix to an output matrix. The minimization is obtained by building a *matching-count matrix*. The entries of the matching-count matrix $MC_{i,j}$ represent the number of entry matches between the input variable number i in the input matrix (X) and the output function number j in the output matrix (Y). It then selects those input-output pairing that gives the maximum matching count, thus minimizing the number of switching operations required. The mapping is achieved by applying a sequence of primitive operations. In MVL this primitive operation is *entry-switching* which is defined as follows:

Entry-Switching: This is denoted by S_{ij}, and it switches the parity of the ijth entry in X. For MVL it is denoted by $S_{ij}(p)$, which means switching the ijth entry of the input matrix to the value p, where $p \in \{0, 1, ..r-1\}$ for r-valued logic.

4.3.1 Output Phase Assignment without complement

The output phase assignment without complement technique proposed by [25] can be explained by considering the following example.

Example

Consider the following mapping in 4-valued logic system

$$X=\begin{bmatrix} 0 & 1 \\ 3 & 2 \\ 1 & 0 \\ 3 & 3 \end{bmatrix} \rightarrow Y=\begin{bmatrix} 3 & 0 \\ 3 & 3 \\ 1 & 1 \\ 0 & 0 \end{bmatrix}$$

The above mapping X $\rightarrow$ Y requires 6 switching operations and the decomposition of this mapping can be written as

$$Y = S_{42}(0)S_{32}(1)S_{22}(3)S_{12}(0)S_{41}(0)S_{11}(3)X.$$

The matching-count matrix of the function will be used to minimize the number of switching operations in MVL system. For the MVL, the entries in the matching-count matrix represent the matching count between an input variable column and an output function phase column which produces the maximum matching count. The different phases of y are obtained as follows

For column y_1

x_1	x_2	$\overrightarrow{y_1}^0$	$\overrightarrow{y_1}^1$	$\overrightarrow{y_1}^2$	$\overrightarrow{y_1}^3$
0	1	3	0	1	2
3	2	3	0	1	2
1	0	1	2	3	0
3	3	0	1	2	3

For column y_2

x_1	x_2	$\overrightarrow{y_2}^0$	$\overrightarrow{y_2}^1$	$\overrightarrow{y_2}^2$	$\overrightarrow{y_2}^3$
0	1	0	1	2	3
3	2	3	0	1	2
1	0	1	2	3	0
3	3	0	1	2	3

The matching-count matrix for the above example is

	y_1	y_2
x_1	(2,0)	(3,0)
x_2	(3,3)	(3,3)

The first co-ordinate of the entry represents the maximum matching count across different phases and the second co-ordinate represents the corresponding phase which produces this matching count. From the above table it is observed that the maximum total matching count is 6 with the following input-output pairing

$$\{(y_1, x_2), (y_2, x_1)\}$$

The mapping X $\rightarrow$ Y after phasing and reordering will be X $\rightarrow$ Y' as follows

$$\text{X}= \begin{bmatrix} 0 & \mathbf{1} \\ 3 & 2 \\ 1 & 0 \\ \mathbf{3} & 3 \end{bmatrix} \quad \rightarrow \quad Y' = \begin{bmatrix} 0 & 2 \\ 3 & 2 \\ 1 & 0 \\ 0 & 3 \end{bmatrix}$$

i.e x_1 is paired with y_2^{-0} (y_1') and x_2 is paired with y_1^{-3} (y_2'). The bold-type entries shown in the X matrix need to be switched to the corresponding entry in Y'. The number of switching operations as can be seen is 2 compared to 6 if the minimization technique is not used. After generating Y' from X, Y is generated from Y' by reordering and phasing as depicted in Figure 4.7 and from the following equations

$$y_1 = y_2^{'-3} = y_2' \ominus 3,$$

$$y_2 = y_1'$$

Since in this technique the entries of the input matrix X are paired with the

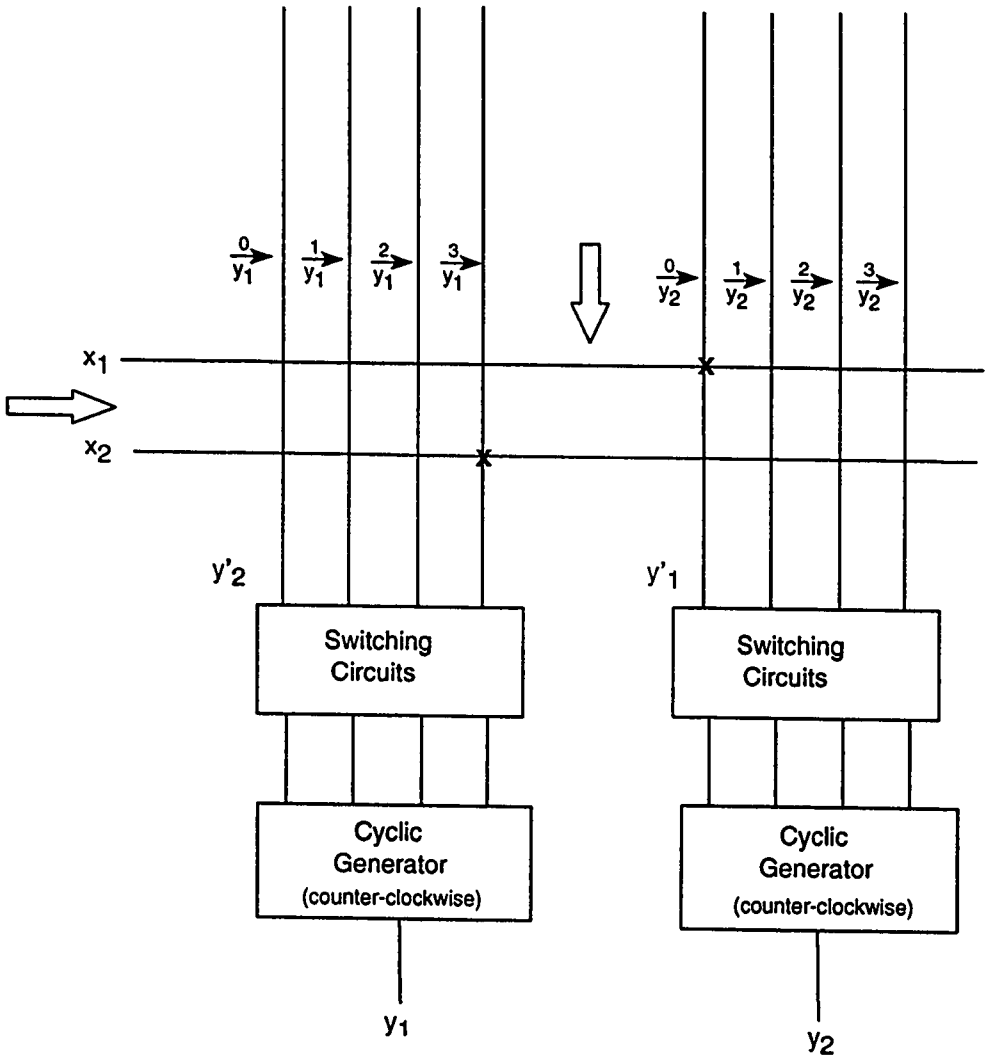


Figure 4.7: Circuit realization of the mapping X $\rightarrow$ Y for output phase without complement.

phase of the output matrix Y, therfore, we may call it **output phase without complement** technique for the purpose of comaprison with other techniques.

4.3.2 Experimental Results

A program has been written to compute a performance measure. The percentage saving in the number of switching elements have been taken as the measure of performance improvement. Table 4.1 depicts the results of experimenting with the output phase without complement technique using 2-input variables 5000 randomly generated 3-valued functions for output phase without complement technique [25]. Table 4.2 and Table 4.3 illustrate the results of similar experiments on 4-valued and 5-valued logic systems respectively. The S_i in the table represent different samples of random functions used in the experiment. It can be observed that percent saving decrease with the increase in radix system.

Table 4.1: The results of applying the algorithm to 2-variables 5000 randomly generated 3-valued functions (Output phase without complement).

Random function	Total no. of switching operations without algorithm	Total no. of switching operations with algorithm	Percentage saving %
S1	26693	14422	45.97
S2	26555	14412	45.73
S3	26816	14812	44.76
S4	27180	15362	43.48
S5	27565	14662	46.81
Average	26962	14734	45.35

Table 4.2: The results of applying the algorithm to 2-variables 5000 randomly generated 4-valued function (Output phase without complement).

Random function	Total no. of switching operations without algorithm	Total no. of switching operations with algorithm	Percentage saving %
S1	29862	17422	41.66
S2	30190	17436	42.25
S3	30664	18536	39.55
S4	29233	17889	38.80
S5	31255	18325	41.37
Average	30241	17922	40.74

Table 4.3: The results of applying the algorithm to 2-variables 5000 randomly generated 5-valued function (Output phase without complement).

Random function	Total no. of switching operations without algorithm	Total no. of switching operations with algorithm	Percentage saving %
S1	32120	20845	35.10
S2	31195	19756	36.67
S3	32163	20838	35.21
S4	29156	18114	37.87
S5	31280	19894	36.40
Average	31183	19889	36.22

Chapter 5

Proposed Decomposition based MVL Synthesis Techniques

In [25], multilevel synthesis of MVL functions has been done using the concept of output phase assignment without complement. There can be other approaches to realize the MVL functions using decomposition techniques. In this thesis we will study and analyze these techniques. The suggested techniques are:

1. Output Phase Assignment with complement.

2. Input Phase Assignment without complement.

3. Input Phase Assignment with complement.

5.1 Output Phase Assignment with complement

In this approach the complement of the output phase assignment is taken for realizing the MVL functions so as to reduce the number of switching operations. In this method we take the complement of the *cyclic* operations of each output function and pair the variables of the input matrix with those function of output matrix (complemented) that gives the maximum matching count.

The complement of a cyclic operation is defined as [4]

$$\overline{x^{-m}} = (r-1) - x^{-m} = \overline{x}^{-m}$$

Example

Let us consider the following mapping in a 4-valued logic function

$$\text{X}= \begin{bmatrix} 1 & 3 \\ 2 & 0 \\ 3 & 2 \\ 0 & 1 \end{bmatrix} \quad \rightarrow \quad Y = \begin{bmatrix} 2 & 3 \\ 1 & 2 \\ 0 & 0 \\ 2 & 1 \end{bmatrix}$$

Let the columns of X be denoted by x_1 and x_2 and the columns of Y be denoted by y_1 and y_2. The different phases of the function column Y with corresponding maximum matching entries with X bold-typed is as follows:

For column y_1

x_1	x_2	$y_1^{\rightarrow 0}$	$y_1^{\rightarrow 1}$	$y_1^{\rightarrow 2}$	$y_1^{\rightarrow 3}$	$\overline{y_1^{\rightarrow 0}}$	$\overline{y_1^{\rightarrow 1}}$	$\overline{y_1^{\rightarrow 2}}$	$\overline{y_1^{\rightarrow 3}}$
1	3	2	3	0	1	**1**	0	3	2
2	0	1	2	3	0	**2**	1	0	3
3	2	0	1	2	3	**3**	2	1	0
0	1	2	3	0	1	1	0	3	2

For column y_2

x_1	x_2	$y_2^{\rightarrow 0}$	$y_2^{\rightarrow 1}$	$y_2^{\rightarrow 2}$	$y_2^{\rightarrow 3}$	$\overline{y_2^{\rightarrow 0}}$	$\overline{y_2^{\rightarrow 1}}$	$\overline{y_2^{\rightarrow 2}}$	$\overline{y_2^{\rightarrow 3}}$
1	3	3	0	1	2	0	**3**	2	1
2	0	2	3	0	1	1	**0**	3	2
3	2	0	1	2	3	3	**2**	1	0
0	1	1	2	3	0	2	**1**	0	3

It is worth noting that if two (or more) phases produce the same (maximum) matching count, any of the two (or more) phases can be taken arbitrarily. The matching-count matrix for this mapping is given as follows:

	$\overline{y_1}$	$\overline{y_2}$
x_1	(3,0)	(2,3)
x_2	(2,2)	(4,1)

As can be seen from the matching-count matrix, all entries are greater than or equal to 1. In general for a mapping of k rows and r-valued logic, the entries of the matching-count have a lower bound of $\lceil(\frac{k}{r})\rceil$. Hence the worst case (upper bound) number of switching operations required for a mapping of X $\rightarrow$ Y of k rows and n columns (assuming X and Y of the same order), in r-valued logic is $\lceil(\frac{k(r-1)}{r})\rceil \times n$ [25]. It can be observed from the matching count matrix that the following pairing gives the maximum matching count:

Pair x_1 with $\overline{y_1^{-0}}$, and pair x_2 with $\overline{y_2^{-1}}$.

The mapping X $\rightarrow$ Y after phasing and reordering will be X $\rightarrow$ Y $'$ as follows:

$$X=\begin{bmatrix} 1 & 3 \\ 2 & 0 \\ 3 & 2 \\ \mathbf{0} & 1 \end{bmatrix} \quad \rightarrow \quad Y' = \begin{bmatrix} 1 & 3 \\ 2 & 0 \\ 3 & 2 \\ 1 & 1 \end{bmatrix}$$

where entries to be switched are shown in bold-type. The number of switching operations required is just 1 as compared to 6 if our technique is not used. The original function Y can be obtained from Y$'$ by reordering and rephasing as depicted in Figure 5.1 and from the following equations

$$y_1 = \overline{y_1'^{-0}}$$

$$y_2 = \overline{y_2'^{-1}}$$

In Figure 5.1, the switching circuit is used to switch the elements of X matrix, so that the input matrix X is directly mapped to output matrix Y. After then, the complement and *counter clockwise* cyclic generator is used to get the original output matrix Y.

5.2 Input Phase Assignment without complement

In [25] the MVL functions have been realized by a decomposition technique that uses only the output phase. It has been observed that if we cycle the input variables and pair them with the output function based on maximum matching-count, the number of switching operations can be minimized. Let us consider an example to illustrate the idea.

Example

Let us consider the following mapping in a 4-valued logic function

$$X=\begin{bmatrix} 1 & 2 \\ 2 & 1 \\ 0 & 0 \\ 3 & 2 \end{bmatrix} \quad \rightarrow \quad Y=\begin{bmatrix} 0 & 2 \\ 1 & 3 \\ 3 & 1 \\ 2 & 3 \end{bmatrix}$$

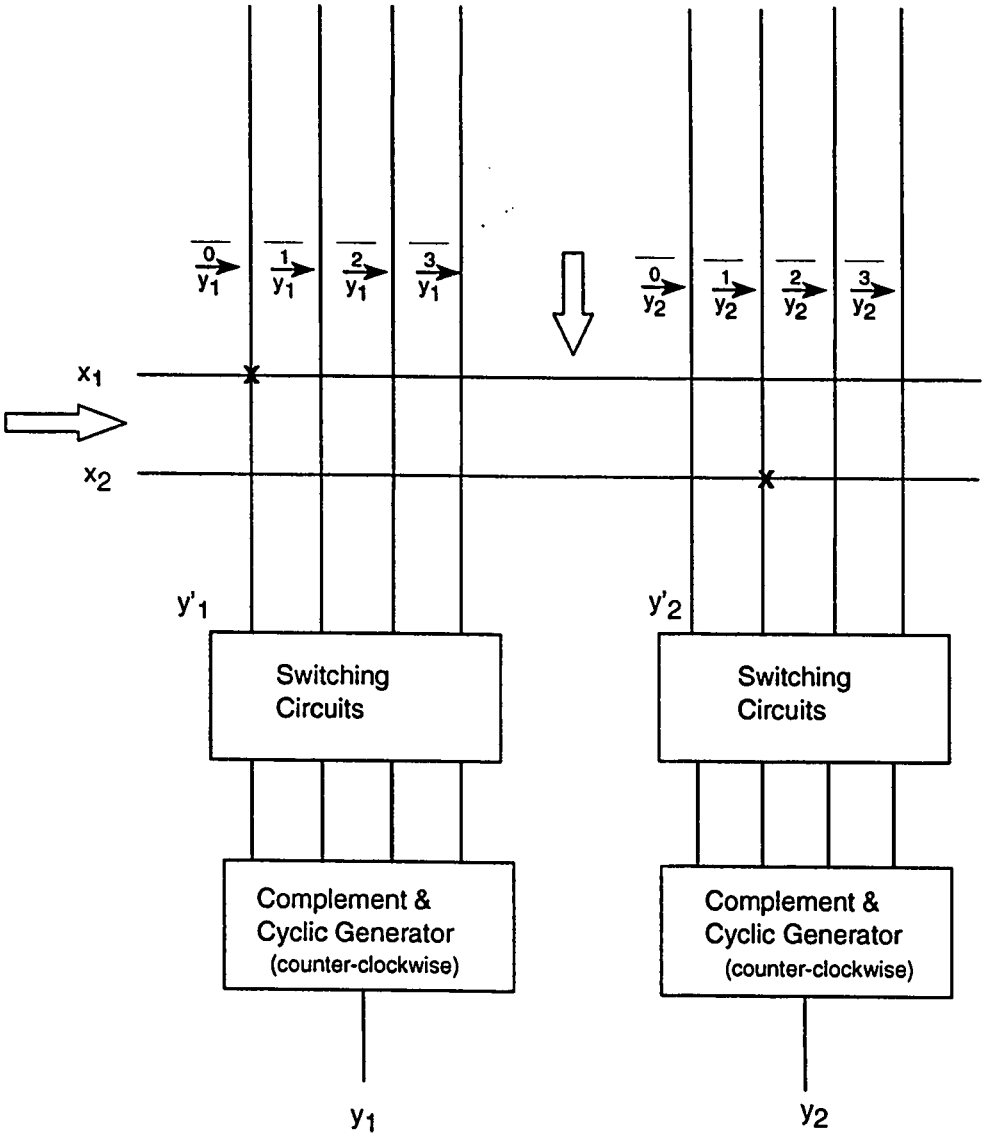


Figure 5.1: Circuit realization of the mapping X $\rightarrow$ Y for output phase with complement.

Let the columns of X be denoted by x_1 and x_2 and the columns of Y be denoted by y_1 and y_2. The different phases of the input column X (bold-typed) with corresponding maximum matching entries with Y is as follows:

For column x_1

y_1	y_2	$\vec{x_1}^0$	$\vec{x_1}^1$	$\vec{x_1}^2$	$\vec{x_1}^3$
0	2	1	2	3	**0**
1	3	2	3	0	**1**
3	1	0	1	2	**3**
2	3	3	0	1	**2**

For column x_2

y_1	y_2	$\vec{x_2}^0$	$\vec{x_2}^1$	$\vec{x_2}^2$	$\vec{x_2}^3$
0	2	2	3	0	**1**
1	3	1	2	3	**0**
3	1	0	**1**	2	**3**
2	3	2	**3**	0	**1**

The matching count matrix for the above example will be

	x_1	x_2
y_1	(4,3)	(2,0)
y_2	(3,1)	(2,1)

In the above matching-count matrix the first co-ordinate indicates the maximum matching count across different phases of X and second co-ordinate indicates the corresponding phase which produces this matching count.

From the matching-count matrix it can be observed that the following pairing gives the maximum matching count:

i.e., pair $\vec{x_1^3}$ with y_1 and pair $\vec{x_2^1}$ with y_2. The mapping X $\rightarrow$ Y after phasing and reordering will be X' $\rightarrow$ Y as follows:

$$X' = \begin{bmatrix} 0 & \mathbf{3} \\ 1 & \mathbf{2} \\ \mathbf{3} & 1 \\ 2 & 3 \end{bmatrix} \quad \rightarrow \quad Y = \begin{bmatrix} 0 & \mathbf{2} \\ 1 & \mathbf{3} \\ \mathbf{3} & 1 \\ 2 & 3 \end{bmatrix}$$

where entries to be switched are shown in bold-type. The number of switching operations as can be seen is just 2 as compared to 7 if our approach is not used. In the input phase assignment case we do not require reordering or phasing to get the original function. Therefore, there is a saving of hardware circuitry required at the output. The circuit realization of the mapping X $\rightarrow$ Y is shown in Figure 5.2. As it can be seen from Figure 5.2, only switching circuit is needed to map directly the input matrix X to the output matrix X.

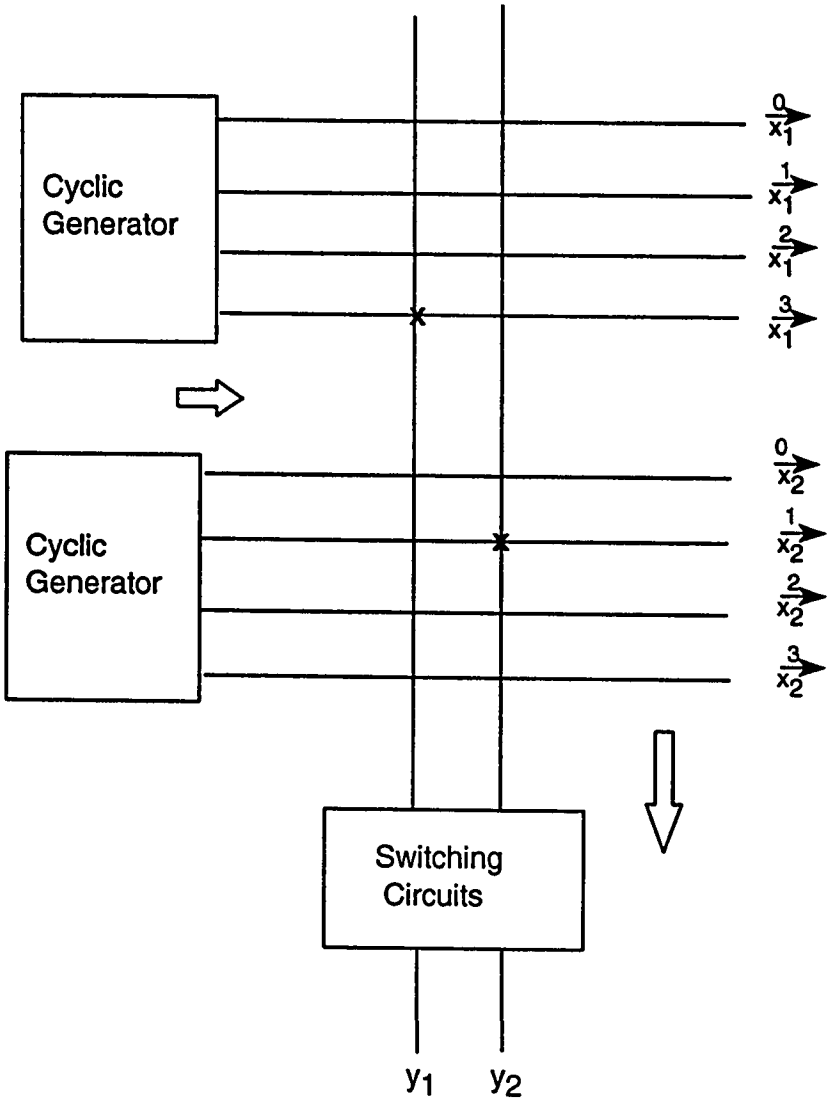


Figure 5.2: Circuit realization of the mapping X $\rightarrow$ Y for input phase without complement.

5.3 Input Phase Assignment with complement

In this approach we take the complement of the input phase assignment for realizing the MVL function. Here we take the complement of the *cyclic* operations of each input variables and pair with the output functions in such a way as to reduce the number of switching operations required.

Example

Let us consider the following mapping in a 4-valued logic function

$$X=\begin{bmatrix}3 & 1\\ 2 & 1\\ 2 & 2\\ 0 & 3\end{bmatrix} \rightarrow Y=\begin{bmatrix}1 & 2\\ 2 & 3\\ 0 & 1\\ 3 & 2\end{bmatrix}$$

The different phases of the input column X (bold-typed) with corresponding maximum matching entries with Y is as follows:

For column x_1

y_1	y_2	$x_1^{\rightarrow 0}$	$x_1^{\rightarrow 1}$	$x_1^{\rightarrow 2}$	$x_1^{\rightarrow 3}$	$\overline{x_1^{\rightarrow 0}}$	$\overline{x_1^{\rightarrow 1}}$	$\overline{x_1^{\rightarrow 2}}$	$\overline{x_1^{\rightarrow 3}}$
1	2	3	0	1	2	0	3	2	1
2	3	2	3	0	1	1	0	3	2
0	1	2	3	0	1	1	0	3	2
3	2	0	1	2	3	3	2	1	0

For column x_2

y_1	y_2	$x_2^{\rightarrow 0}$	$x_2^{\rightarrow 1}$	$x_2^{\rightarrow 2}$	$x_2^{\rightarrow 3}$	$\overline{x_2^{\rightarrow 0}}$	$\overline{x_2^{\rightarrow 1}}$	$\overline{x_2^{\rightarrow 2}}$	$\overline{x_2^{\rightarrow 3}}$
1	2	1	2	3	0	2	1	0	3
2	3	1	2	3	0	2	1	0	3
0	1	2	3	0	1	1	0	3	2
3	2	3	0	1	2	0	3	2	1

The matching count matrix for the above example will be

	$\overline{x}_1$	$\overline{x}_2$
y_1	(2,3)	(3,1)
y_2	(2,2)	(2,0)

In the above matching-count matrix the first co-ordinate indicates the maximum matching count across different phases of X (complement) and second co-ordinate indicates the corresponding phase which produces this matching count.

It has been observed from the above matching count matrix that the following pairing gives the maximum matching count which in turn reduces the switching operations.

i.e., pair $\overline{x_1^{\rightarrow 2}}$ with y_2 and pair $\overline{x_2^{\rightarrow 1}}$ with y_1. The mapping X $\rightarrow$ Y after phasing and reordering will be X$'$ $\rightarrow$ Y as follows

$$X' = \begin{bmatrix} 1 & 2 \\ 1 & 3 \\ 0 & 3 \\ 3 & 1 \end{bmatrix} \quad \rightarrow \quad Y = \begin{bmatrix} 1 & 2 \\ 2 & 3 \\ 0 & 1 \\ 3 & 2 \end{bmatrix}$$

where bold-type entries need to be switched in order to accomplish the mapping. The number of switching operations required is just 3 as compared to 7 required without using this approach. The circuit realization of the mapping X $\rightarrow$ Y is shown in Figure 5.3. Here also we do not need any conversion circuitry to get the original function back as in the case of input phase without complement assignment.

5.4 Experimental Results

Experiments to test the effectiveness of the proposed technique in reducing the number of switching operations required for MVL function for different approaches has been carried out. A program has been written in C. It accepts different randomly generated functions and synthesizes them for different radix and different synthesis techniques. Then the program applies the proposed technique and computes a performance measure. The percentage saving in the number of switching elements has been taken as a measure of performance improvement. In this comparison the same sample of random functions have been used for every technique.

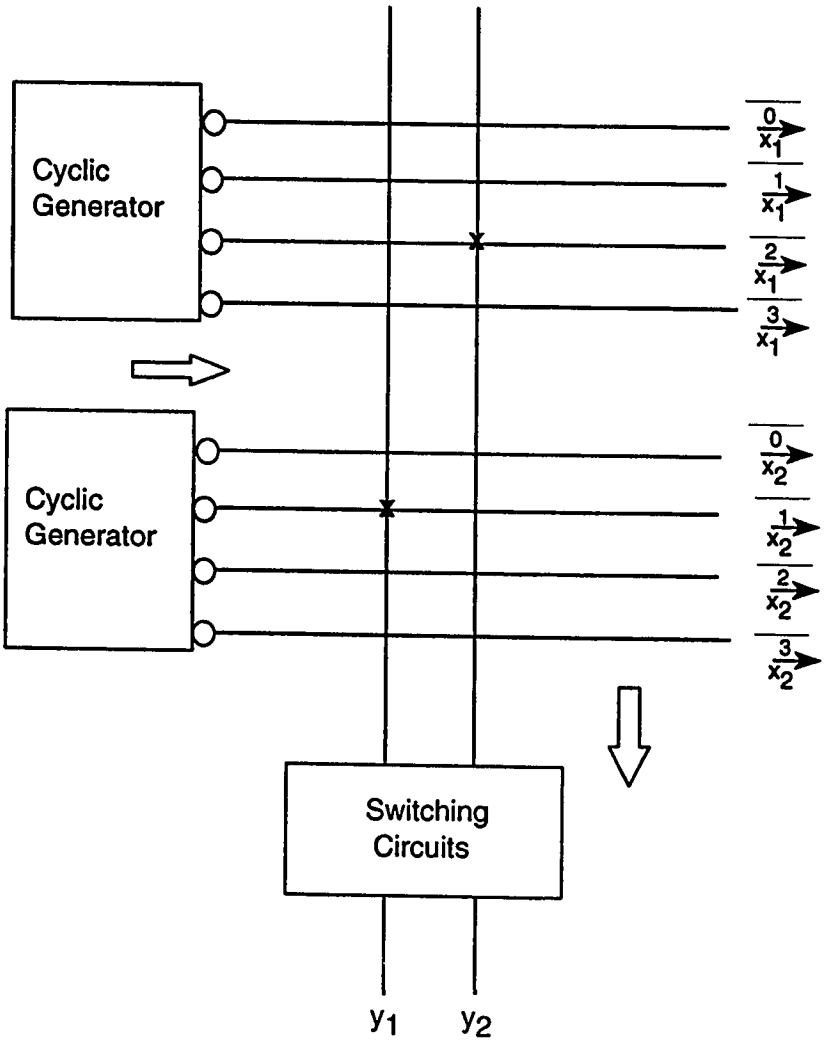


Figure 5.3: Circuit realization of the mapping X → Y for input phase with complement.

Tables 5.1, 5.2, and 5.3 depict the results for output phase with complement technique by applying the same matching count matrix methodology. From the results it is observed that, on the average, the output phase with complement technique performs slightly better than output phase without complement technique.

The input phase without complement technique has been applied to the same sample of 2-input variables 5000-randomly generated functions. These results are shown in Tables 5.4, 5.5 and 5.6 for radix 3,4 and 5 respectively.

Similar experiments have been carried out for input phase with complement technique. The results are shown in Tables 5.7, 5.8 and 5.9 for radix 3,4 and 5 respectively. From the tables it is evident that the complement of input phase performs slightly better than the input phase without complement technique.

It can be observed from the results of the above four mentioned techniques, that on the average input phase with complement techniques gives better percentage of saving of switching operations as compared to the other three techniques. But for realizing the complement technique (both for input phase and output phase), the *complement of cyclic* operation is required. The implementation of these *complement of cyclic* operation is advantageous only if input variables and their complements are available. If it is assumed that the complement of an input variable is available then the cost of realizing the *cycle* operator and the *complement of cycle* operator is the same [31] and as such *complement* techniques provide better solution than the it without complement techniques.

It can also be observed from the results that, on average, input phase techniques (complement and without complement) give better performance compared to output phase techniques in terms of the switching operations. Furthermore, both input phase techniques do not require any *counter-clockwise cyclic* operation to get the original output matrix. Thus input phase techniques have less hardware complexity as compared to output phase techniques.

A clear observation that can be drawn from these results is that percent saving decreases with the increase in radix system. This is due to the fact that the lower bound on the entries of the matching-count matrix is $\lceil(\frac{k}{r})\rceil$. Therefore, as the radix increases the lower bound decreases and it results in more number of switching operations required.

Table 5.1: The results of applying the algorithm to 2-variables 5000 randomly generated 3-valued functions (Output phase with complement).

Random function	Total no. of switching operations without algorithm	Total no. of switching operations with algorithm	Percentage saving %
S1	26693	14302	46.42
S2	26555	14377	45.86
S3	26816	14572	45.66
S4	27180	15413	43.3
S5	27565	14715	46.62
Average	26962	14676	45.57

Table 5.2: The results of applying the algorithm to 2-variables 5000 randomly generated 4-valued functions (Output phase with complement).

Random function	Total no. of switching operations without algorithm	Total no. of switching operations with algorithm	Percentage saving %
S1	29862	17143	42.59
S2	30190	17324	42.62
S3	30664	18592	39.37
S4	29233	17482	40.20
S5	31255	17998	42.42
Average	30241	17708	41.44

Table 5.3: The results of applying the algorithm to 2-variables 5000 randomly generated 5-valued functions (Output phase with complement).

Random function	Total no. of switching operations without algorithm	Total no. of switching operations with algorithm	Percentage saving %
S1	32120	19922	37.98
S2	31195	19120	38.71
S3	32163	20134	37.40
S4	29156	18345	37.08
S5	31280	19228	38.53
Average	31183	19350	37.95

Table 5.4: The results of applying the algorithm to 2-variables 5000 randomly generated 3-valued functions (Input phase without complement).

Random function	Total no. of switching operations without algorithm	Total no. of switching operations with algorithm	Percentage saving %
S1	26693	13905	47.91
S2	26555	14312	46.10
S3	26816	14492	45.96
S4	27180	14962	44.95
S5	27565	14308	48.1
Average	26962	14395	46.61

Table 5.5: The results of applying the algorithm to 2-variables 5000 randomly generated 4-valued functions (Input phase without complement).

Random function	Total no. of switching operations without algorithm	Total no. of switching operations with algorithm	Percentage saving %
S1	29862	16955	43.22
S2	30190	16998	43.70
S3	30664	17998	41.31
S4	29233	16678	42.95
S5	31255	17455	44.15
Average	30241	17217	43.08

Table 5.6: The results of applying the algorithm to 2-variables 5000 randomly generated 5-valued functions (Input phase without complement).

Random function	Total no. of switching operations without algorithm	Total no. of switching operations with algorithm	Percentage saving %
S1	32120	19655	38.80
S2	31195	18782	39.57
S3	32163	19356	39.82
S4	29156	17345	40.51
S5	31280	18442	41.04
Average	31183	18716	39.98

Table 5.7: The results of applying the algorithm to 2-variables 5000 randomly generated 3-valued functions (Input phase with complement).

Random function	Total no. of switching operations without algorithm	Total no. of switching operations with algorithm	Percentage saving %
S1	26693	13746	48.50
S2	26555	14121	46.82
S3	26816	14413	46.25
S4	27180	14913	45.13
S5	27565	14435	47.63
Average	26962	14326	46.86

Table 5.8: The results of applying the algorithm to 2-variables 5000 randomly generated 4-valued functions (Input phase with complement).

Random function	Total no. of switching operations without algorithm	Total no. of switching operations with algorithm	Percentage saving %
S1	29862	16910	43.37
S2	30190	16824	44.27
S3	30664	17855	41.77
S4	29233	16850	42.36
S5	31255	17120	45.22
Average	30241	17112	43.41

Table 5.9: The results of applying the algorithm to 2-variables 5000 randomly generated 5-valued functions (Input phase with complement).

Random function	Total no. of switching operations without algorithm	Total no. of switching operations with algorithm	Percentage saving %
S1	32120	19112	40.50
S2	31195	18165	41.77
S3	32163	19148	40.46
S4	29156	17520	39.91
S5	31280	17845	42.95
Average	31183	18358	41.13

Chapter 6

Output Phase Optimization of MVL functions

Several techniques for the minimization of sum-of-product expression in binary circuits has been presented in the past. These have been studied in detail in [15]. The minimization techniques for the sum-of-product expression in multiple-valued logic have been investigated by many researchers. For example in [26],[17], and [49] the optimal output encoding and minimization techniques have been presented.

In this chapter optimal output phase assignment to minimize the number of implicants in a sum-of-product expression is presented. Here the sum refers to the *tsum* operator and product refers to the *min* operation on functions of the input variables which are *literals*.

6.1 Previous Output Phase Permuted Technique

In [32], an output phase optimization technique has been presented. Here, the optimal assignment is done by minimizing all the $r!$ functions for a given r-valued function. Let us call this technique the output phase *permuted* technique. There are $r!$ different ways of assigning for r-valued single output function. The function which requires the minimum number of implicants among $r!$ functions is selected. This function is called the optimal output permuted function. The above technique can be explained by considering the following example.

Example

Consider a 3-valued function $f(= f_1)$ as shown in Figure 6.1 [32]. There are $3! = 6$ different possible permutations of the ternary values 0, 1 and 2. These permutations are $p_1 = 012$, $p_2 = 021$, $p_3 = 120$, $p_4 = 102$, $p_5 = 201$, and $p_6 = 210$. As an example, the function f_2 is obtained from f_1 by permuting the truth values as follows

$$0 \Rightarrow 1$$

$$1 \Rightarrow 0$$

$$2 \Rightarrow 2$$

After applying all permutations p_i to function $f(= f_1)$, 6 different functions are obtained which are shown in Figure 6.1. Then the Dueck & Miller heuristic provided by HMALET software has been used to minimize all these functions. The number

of implicants needed to cover each function f_i are shown inside angle-bracket <>. The number of implicants shown against *user* are the number of implicants required without using the minimization heuristic. The implicants shown with *DM*, are the implicants which are obtained after applying the DM heuristic. The original function f_1 can be represented in the sum-of-product form as

$$f = (1[^2\{x_1\}^2]) \bullet (1[^0\{x_2\}^2]) \uplus (1[^2\{x_1\}^2]) \bullet (1[^1\{x_2\}^1]) \uplus (1[^0\{x_1\}^0]) \bullet (1[^2\{x_2\}^2])$$
$$\uplus (2[^0\{x_1\}^0]) \bullet (2[^0\{x_2\}^0]) \uplus (2[^1\{x_1\}^1]) \bullet (2[^2\{x_2\}^2])$$

Thus the original function f_1 require 5 implicants. After minimizing all the *permuted* function, it can be observed that only 3 implicants are needed to cover the function f_3. Thus f_3 is selected, and it is termed as the optimal output permuted function. The optimal function f_3 is represented in the sum-of-product form as

$$f_{opt} = (1[^0\{x_1\}^2]) \bullet (1[^1\{x_2\}^1]) \uplus (1[^1\{x_1\}^1]) \bullet (1[^2\{x_2\}^2]) \uplus (1[^0\{x_1\}^1]) \bullet (1[^0\{x_2\}^1])$$

The original function can be realized by adding an unary operation to the optimal output permuted function.

Although this technique results in less number of implicants needed to cover a given function, but the disadvantage is that for each function, it requires $r!$ permutations. For higher radices this technique is particularly not suitable because as the radix increases the number of possible permuted functions also increases. For example, in 6-valued system, the number of permuted functions for each function is 6!=720 i.e, in order to find the minimum number of implicants for a given function, one has to minimize all the 720 possible permuted functions.

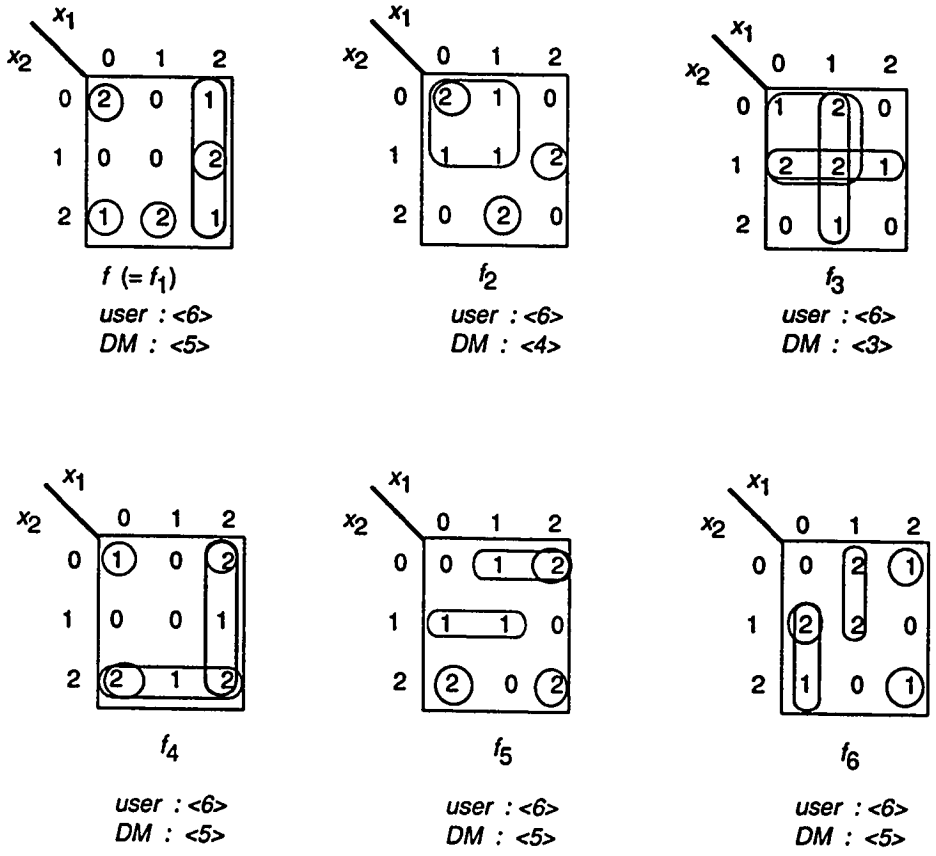


Figure 6.1: All functions obtained by permuting the ternary values.

6.2 A Proposed Output Phase Cyclic Technique

For a given n-variable r-valued function $f(X)$, there can be a maximum of $(r-1)$ *cyclic* functions. These cyclic functions are denoted by $f_1, f_2, \cdots, f_{r-1}$. These $(r-1)$ cyclic functions are obtained by performing the *cyclic* operation on the given function f. Next, all these cyclic functions are minimized. The function with the least number of implicants is then selected. We call this function the optimal output *cyclic* function denoted by $f_{opt}^{\rightarrow i}$. The original output function f is obtained from optimal output *cyclic* function $f_{opt}^{\rightarrow i}$ by performing the *counter clockwise cyclic* operation D on it such that $D(f_{opt}^{\rightarrow i}) = f$ holds. This technique is termed as the output phase *cyclic* technique.

Example 1

Consider a 3-valued function $f(= f_1)$ as shown in Figure 6.2. This function is similar to the function as shown in Figure 6.1. There are $(r-1) = 2$ possible cyclic functions, these are denoted by f_2 and f_3. The DM heuristic is then applied to the original function and each *cycled* function. The number inside <> shows the number of implicants needed to cover the individual function. After minimization, the original function f_1 require 5 implicants, whereas the function $(f_3 = f^{\rightarrow^2})$ requires the least number of implicants i.e., 3 implicants. Thus the function f_3 is called the optimal output *cyclic* function denoted by $f_{opt}^{\rightarrow^2}$. The original function f is obtained by the

relation :

$$D(f_{opt}^{\rightarrow 2}) = f_{opt}^{\leftarrow 2} = f_{opt}^{\rightarrow 1} = f$$

Thus it is observed that only 3 functions are required to minimize in order to obtain the optimal output function as compared to minimization of 6 functions required by [32]. Also the realization of the unary operation to get back the original function has not been mentioned in [32], whereas, in our approach the original function can be obtained easily by just introducing a *counter clockwise* cyclic operator at the output. The realization of the *counter clockwise* operator has been discussed in Section 3.2.1.

Another advantage of our *cyclic* approach is that we need to minimize only r number of functions to select the optimal function, whereas $r!$ functions are needed to be minimized by the output phase *permuted* technique as mentioned in [32]. An analysis of the number of minimization required by our output phase *cyclic* method and the output phase *permuted* technique by [32] is shown in Table 6.1. It can be observed that the number of functions to be minimized by output phase *permuted* technique increases drastically with the increase in radix, whereas the number of minimization required by our approach is a function of the radix. For example, for 6-valued function we need to minimize only 6 functions to get the optimal output function as compared to 720 functions which are needed to be minimized by output phase *permuted* technique.

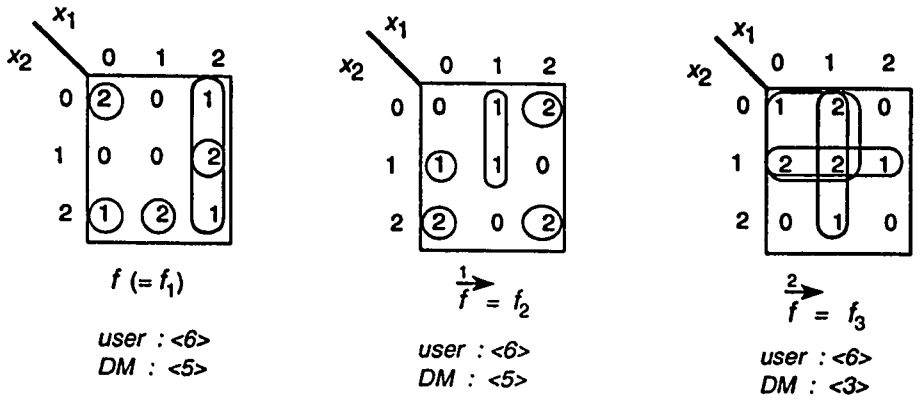


Figure 6.2: All functions obtained by *cycling* the function f.

Table 6.1: Number of functions required by the output phase *cyclic* and the ouput phase *permuted* technique.

Radix	No. of functions required by	
r	Output *cyclic* technique	Output *permuted* technique
3	3	6
4	4	24
5	5	120
6	6	720
7	7	5040

Example 2

Consider a 2-variable 4-valued *increasing* function f as shown in Figure 6.3. The different phases of the output functions are also shown in the same figure. After applying the DM heuristic, the original function is represented in the sum-of-product form as

$$f = (1[^1\{x_1\}^1]) \bullet (1[^0\{x_2\}^1]) \uplus (1[^0\{x_1\}^0]) \bullet (1[^1\{x_2\}^1]) \uplus (2[^2\{x_1\}^3]) \bullet (2[^0\{x_2\}^1])$$
$$\uplus (2[^0\{x_1\}^1]) \bullet (2[^2\{x_2\}^2]) \uplus (3[^0\{x_1\}^3]) \bullet (3[^3\{x_2\}^3] \uplus (3[^2\{x_1\}^3]) \bullet (3[^2\{x_2\}^2])$$

Thus the original function require 6 implicants. The cyclic operation is then applied to the original function f. We can observe that all the three cycled functions f_2, f_3 and f_4 require 5 implicants. Among these, the function which is generated first is taken as the optimal output function. Therefore, $f^{\rightarrow^1}$ is the optimal output function. This function is represented in the sum-of-product form as

$$f_{opt}^{\rightarrow^1} = (1[^0\{x_1\}^0]) \bullet (1[^1\{x_2\}^0]) \uplus (2[^1\{x_1\}^1]) \bullet (2[^0\{x_2\}^1]) \uplus (2[^0\{x_1\}^0]) \bullet (2[^1\{x_2\}^1])$$
$$\uplus (3[^0\{x_1\}^1]) \bullet (3[^2\{x_2\}^2]) \uplus (3[^2\{x_1\}^3]) \bullet (3[^0\{x_2\}^1])$$

Example 3

Figure 6.4 shows the output cycled functions for a 2-variable 4-valued *decreasing* function. The original function ($f = f_1$) is represented in the sum-of-product form as

$$f = (1[^0\{x_1\}^2]) \bullet (1[^0\{x_2\}^2]) \uplus (1[^1\{x_1\}^1]) \bullet (1[^0\{x_2\}^0]) \uplus (2[^0\{x_1\}^0]) \bullet (2[^0\{x_2\}^0])$$

Thus the original function f requires 3 implicants to express the function after applying the DM heuristic. The *cyclic* operation is then applied to the original *de-*

creasing function. The DM heuristic is then applied to each function. It is observed that the cycled output functions f_2 and f_3 require 5 implicants each, whereas f_4 requires 4 implicants. Thus, the 2-variable *decreasing* function performs worse than the original function by employing the output phase *cyclic* technique.

From the above examples, it is seen that *increasing* functions perform better using the proposed output phase *cyclic* technique, whereas, the *decreasing* functions require more implicants using the same technique. However, the *decreasing* function can be transformed into *increasing* functions by taking their *complements* as shown in Figure 6.5. The original function f after applying DM heuristic requires 5 implicants. The cycled functions f_2 and f_3 require 4 implicants, whereas f_4 require 5 implicants. The function f_2 is taken as the optimal solution. Thus, our proposed approach performs effectively for both *increasing* and *decreasing* functions.

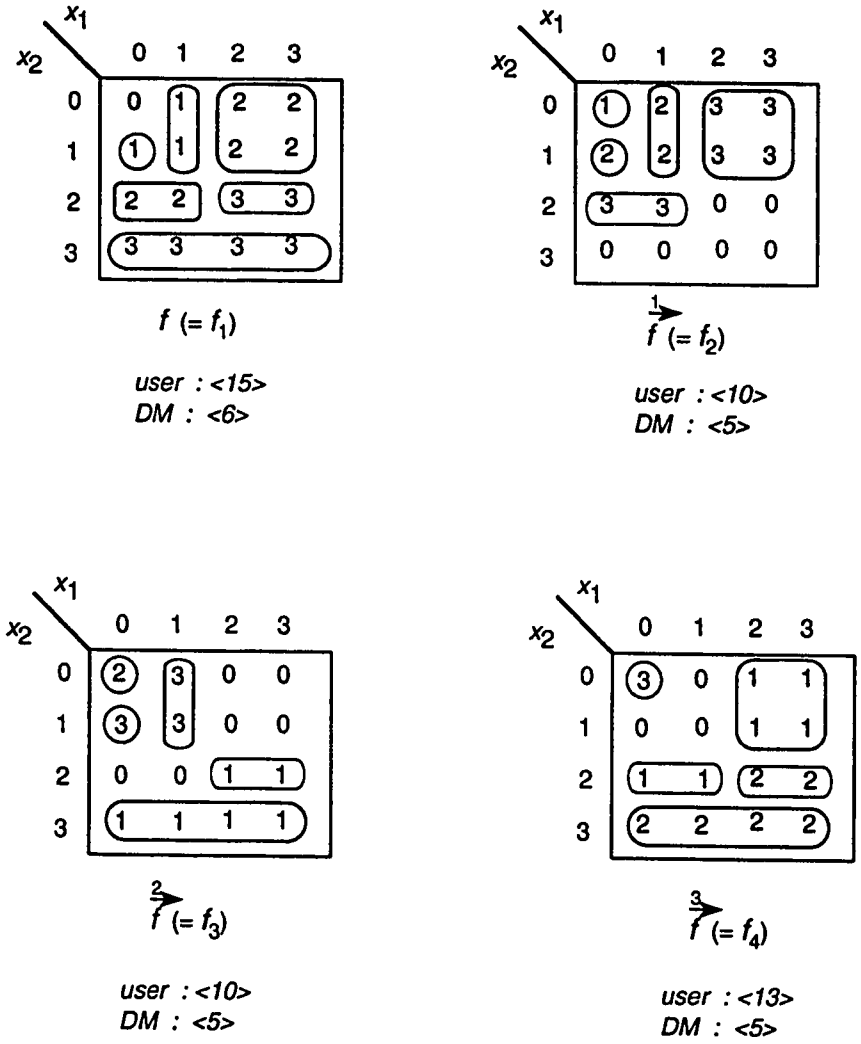


Figure 6.3: The map representation of output functions for Example 2.

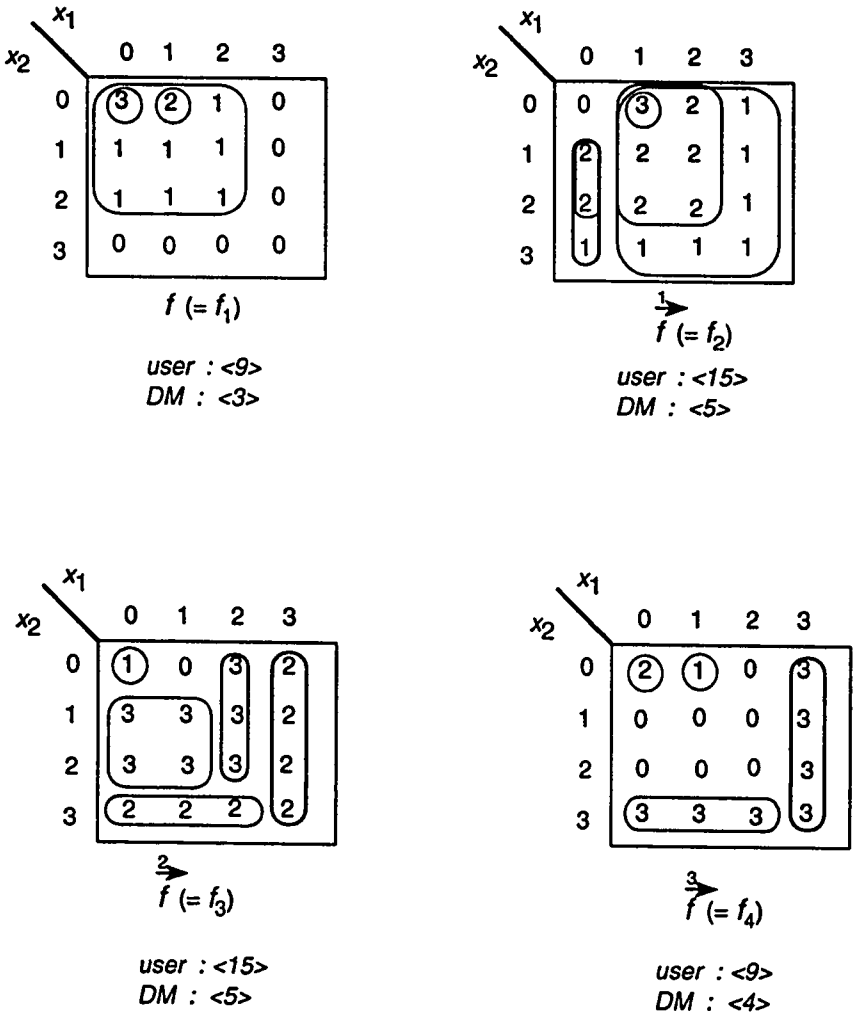


Figure 6.4: The map representation of output functions for Example 3.

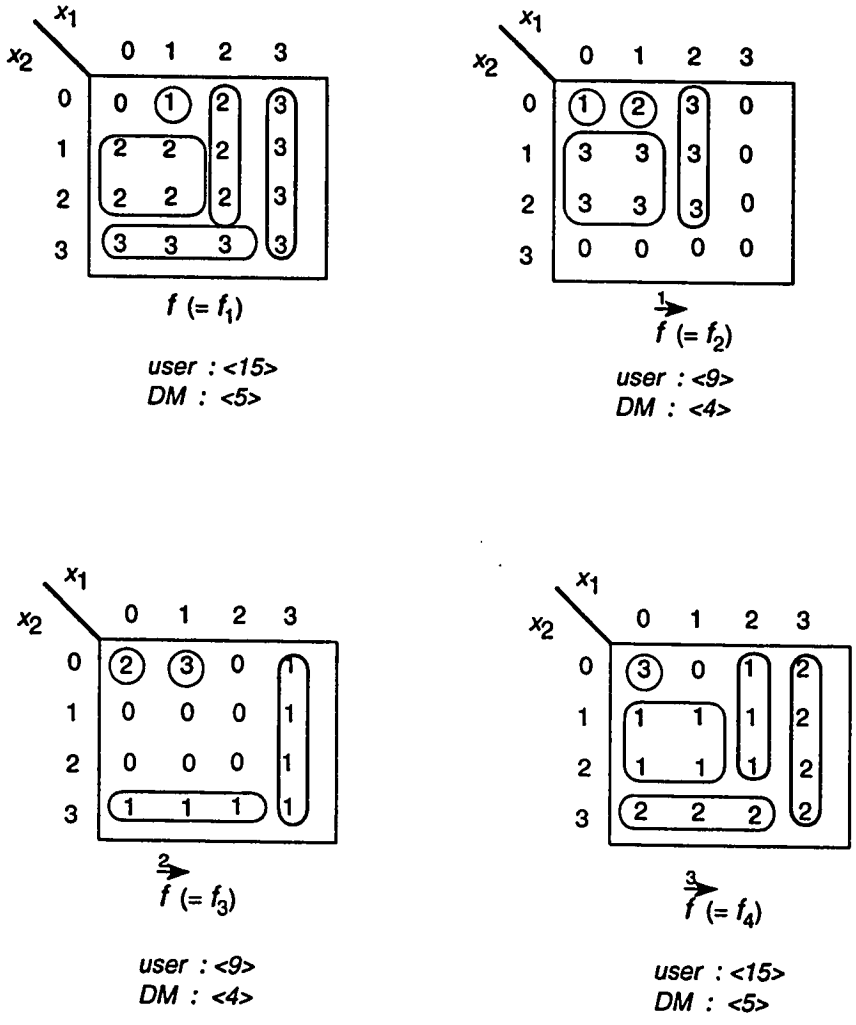


Figure 6.5: The map representation of *complement* of *decreasing* functions for Example 3.

6.3 Experimental Results

Experiments to evaluate the effectiveness of the proposed output phase *cyclic* technique have been carried out. The programming has been done in C language under Unix operating system. The CAD tool, HAMLET (Heuristic Analyzer for Multiple-Valued Logic Expression Translation) installed on the Sun workstations, is used for the multi-valued logic minimization. The HAMLET includes implementation for both Pomper and Armstrong (PA) and Dueck and Miller (DM) methods.

The main steps of the program for evaluating the effectiveness of the output phase *cyclic* technique has been shown in Figure 6.6. In the first step different classes of functions have been generated. Then the cycled output functions are obtained by performing the *cyclic* operation on each output function. A translator program has been written which converts the representation of each output functions to a form to be acceptable to the HAMLET. The HAMLET software then applies the Dueck & Miller and Pomper & Armstrong minimization heuristics to each original function f and the output cycled functions $f^{\rightarrow^i}$ $(i = 1, 2 \cdots, r-1)$ to obtain the minimal number of implicants required by each output functions. Among the cycled output functions, the function with the minimum number of implicants is selected. This function is called the optimal output *cyclic* function, represented by $f_{opt}^{\rightarrow^i}$. The average number of implicants are then computed for the original functions f and the optimal output *cyclic* functions $f_{opt}^{\rightarrow^i}$. The percentage saving in terms of the number

of implicants is then calculated on the average for all functions.

For the purpose of analysis, different classes of one-variable and two-variable functions with different radices have been taken into account. Because of the simplicity, one variable functions are taken as the starting point for evaluating the effectiveness of the proposed output phase *cyclic* technique.

The first measure of performance is the average number of implicants required by the original function f and the optimal output *cyclic* function $f_{opt}^{\rightarrow i}$. In order to compute the percentage saving in the number of implicants, the minimum of the average implicants obtained by DM and PA heuristic is taken as the reference. Table 6.2 shows the percentage saving for all one-variable functions for radix 3,4,5 and 6. It can be observed that our proposed technique gives a performance improvement in terms of the average number of implicants for the considered radices. The proposed output phase *cyclic* technique also performs better consistently using both the DM and PA heuristics. It can be observed that PA and DM heuristic gives the same percentage saving for radix 3. However, PA approach performs better than DM heuristic for radix 4,5 and 6. Similar analysis has been carried out for one-variable *increasing*, *decreasing* and *mixed* functions as shown in Table 6.3, Table 6.4 and Table 6.5 respectively. Both the one-variable *increasing* and *decreasing* functions give almost the same percentage saving. It can also observed that the PA heuristic always performs equal or better than DM heuristic for both these functions. The PA heuristic provides a minimal expression obtained by successively selecting prime

implicants.Since for unary functions there can be at most one prime implicant associated with each minterm, therefore, the PA heuristic performs better than the DM heuristic. Thus, on the average, for one-variable r-valued functions PA heuristic always performs as good as or better than the DM heuristic. However, for one-variable *mixed* function, DM heuristic is slightly better than PA heuristic for radix 6, while for other radices PA performs better.

The same experiment has been repeated for 2-variable r-valued functions. 3000 random functions were generated and then output *cyclic* functions were obtained. Table 6.6 shows the results after applying the DM and PA heuristics. It is observed that the DM heuristic consistently gives better percentage saving than the PA heuristic. But as compared to one-variable functions, the percentage saving decreases. Table 6.7 shows the results for 2-variable *increasing* functions. It can be observed that these functions give significant percentage saving in terms of the average number of implicants by applying our proposed technique. It is also observed that for all the radices considered, PA heuristic performs better than the DM heuristic. Similarly, the results for 2-variable *decreasing* functions are shown in Table 6.8. The results show that there is a very negligible percentage saving for radices 3,4, and 5. For radix 6 there is a small saving in average implicants using DM heuristic whereas by PA heuristic the performance deteriorates and it does not provide any saving. Table 6.9 depicts the results for two-variable *symmetric* functions. It can be observed that these functions give a reasonable percentage saving by both DM and

PA heuristic.

Another measure of performance of the proposed output phase *cyclic* technique can be made with respect to the number of output *cyclic* functions which perform better, equal or worse than the original functions. The term "Better" in table is used to denote the number of functions which require less number of implicants than the original functions. Similarly, the terms "Equal" and "Worse" are used for the number of functions which require equal or more number of implicants than the original functions, respectively. The term "Percentage Success" used in tables indicates the percentage of output *cyclic* functions which perform equal or better than the original functions. In Table 6.10 all the one-variable functions for different radices are considered. It is observed that for radix 3 both the DM and PA heuristic gives the same percentage of success. Although PA heuristic gives better performance as compared to DM heuristic in terms of the number of functions which require less number of implicants for radices 4,5 and 6, however, overall DM gives better percentage of success. Table 6.11 and Table 6.12 show the results for one-variable *increasing* and *decreasing* functions respectively. It can be observed that almost 100% percentage success is obtained for both these types of functions using our approach. DM and PA heuristic almost give the same percentage success, however, only for radix 5 and 6, PA heuristic give some functions which require more implicants than the original functions. Here also PA heuristic provide equal or more number of better functions (i.e, functions require less implicants) as com-

pared to DM heuristic, but overall by DM heuristic, all the optimal cyclic functions require equal or smaller number of implicants than the original functions for all the radices considered. Table 6.13 shows the resulted for one-variable *mixed* functions. Although these functions give a better percentage of success, but their performance is slightly inferior than the *increasing* and *decreasing* functions. Here also the overall performance of DM is equal or slightly better than the PA heuristic.

Similar experiments have been carried out for 2-variable functions. Table 6.14 shows the result for 3000 randomly generated two-variable functions. For these types of functions PA heuristic provide more number of functions which perform better than the original functions as compared to DM heuristic for radix 4,5 and 6. However, the overall better percentage success is obtained by DM heuristic. The results for 2-variable *increasing* and *decreasing* functions are shown in Table 6.15 and Table 6.16 respectively. It can be observed that the performance of 2-variable *increasing* functions is much better than the 2-variable *decreasing* functions. *Increasing* functions gives percentage success almost for 99 % of the functions. For *increasing* functions, the DM heuristic give better percentage success as compared to PA heuristic. Also for *increasing* functions, DM heuristic give more functions which require less number of implicants than the original functions. The *decreasing* functions give a smaller percentage of success as compared to *increasing* functions. However, the *decreasing* functions can be converted into *increasing* functions by taking the complement of *decreasing* functions. It is also observed that for *decreasing*

functions PA heuristic give better percentage success and provide more "better" functions as compared to DM heuristic.

Table 6.17 depicts the results for 2-variable *symmetric* functions. It can be observed that our proposed technique give a substantial percentage success for *symmetric* functions also. We get slightly more "better" functions by PA heuristic, but the overall better percentage success is obtained by DM heuristic.

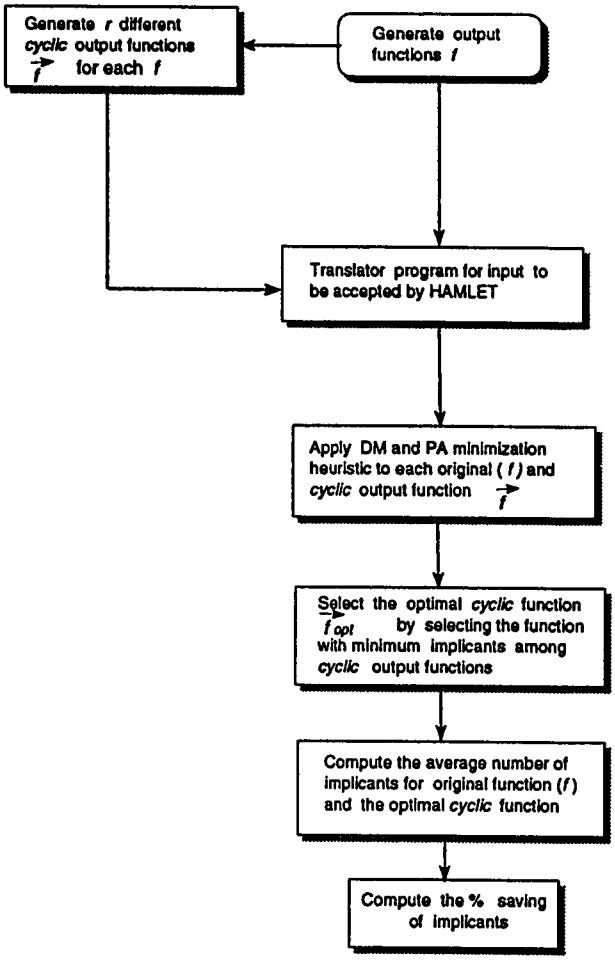


Figure 6.6: Main steps of the program for evaluating the effectiveness of the proposed technique.

Table 6.2: All one variable functions.

Avg. # of implicants	Radix 3		Radix 4		Radix 5		Radix 6	
	DM	PA	DM	PA	DM	PA	DM	PA
f	1.63	1.63	2.29	2.33	3.10	3.12	3.98	3.85
$f_{opt}^{\rightarrow}$	1.25	1.25	1.76	1.75	2.43	2.41	3.14	3.12
% Saving	23.07%	23.07%	22.92%	23.61%	20.64%	21.14%	18.26%	18.76%

Table 6.3: One variable *increasing* functions.

Avg. # of implicants	Radix 3		Radix 4		Radix 5		Radix 6	
	DM	PA	DM	PA	DM	PA	DM	PA
f	1.43	1.43	1.84	1.84	2.28	2.26	2.75	2.74
$f_{opt}^{\rightarrow}$	1.14	1.14	1.45	1.45	1.85	1.85	2.30	2.26
% Saving	20.00%	20.00%	21.05%	21.05%	18.24%	18.24%	15.93%	17.45%

Table 6.4: One variable *decreasing* functions.

Avg. # of implicants	Radix 3		Radix 4		Radix 5		Radix 6	
	DM	PA	DM	PA	DM	PA	DM	PA
f	1.43	1.43	1.84	1.84	2.28	2.28	2.75	2.72
$f_{opt}^{\rightarrow}$	1.14	1.14	1.45	1.45	1.85	1.82	2.30	2.23
% Saving	20.00%	20.00%	21.05%	21.05%	18.84%	19.23%	15.60%	18.00%

Table 6.5: One variable *mixed* functions.

Avg. # of implicants	Radix 3		Radix 4		Radix 5		Radix 6	
	DM	PA	DM	PA	DM	PA	DM	PA
f	1.90	1.90	2.43	2.49	3.13	3.20	3.87	4.00
$f_{opt}^{\rightarrow}$	1.40	1.40	1.86	1.86	2.55	2.55	3.14	3.16
% Saving	26.32%	26.32%	23.38%	23.38%	18.27%	18.27%	18.80%	18.20%

Table 6.6: Randomly generated two variable functions.

Avg. # of implicants	Radix 3		Radix 4		Radix 5		Radix 6	
	DM	PA	DM	PA	DM	PA	DM	PA
f	3.71	3.71	7.27	7.47	12.25	12.84	18.54	19.68
$f_{opt}^{\rightarrow}$	3.19	3.18	6.47	6.54	11.10	11.37	16.96	17.68
% Saving	14.10%	14.24%	10.90%	9.94%	9.44%	7.20%	8.54 %	4.65%

Table 6.7: Randomly generated two variable *increasing* functions.

Avg. # of implicants	Radix 3		Radix 4		Radix 5		Radix 6	
	DM	PA	DM	PA	DM	PA	DM	PA
f	2.79	2.76	4.22	4.20	4.01	3.97	4.71	4.69
$f_{opt}^{\rightarrow}$	1.83	1.81	2.89	2.87	2.83	2.80	3.51	3.49
% Saving	33.47%	34.17%	30.88%	31.43%	28.84%	29.48%	25.15%	25.54%

Table 6.8: Randomly generated two variable *decreasing* functions.

Avg. # of implicants	Radix 3		Radix 4		Radix 5		Radix 6	
	DM	PA	DM	PA	DM	PA	DM	PA
f	1.84	1.84	2.20	2.19	3.72	3.83	4.66	4.81
$f_{opt}^{\rightarrow}$	1.75	1.74	2.18	2.15	3.65	3.69	4.59	4.67
% Saving	5.02%	5.17%	0.49%	1.84%	1.96%	0.81%	1.57%	-0.21%

Table 6.9: Randomly generated 2-variable *symmetric* functions.

Avg. # of implicants	Radix 3		Radix 4	
	DM	PA	DM	PA
f	4.42	4.40	7.84	8.00
$f_{opt}^{\rightarrow}$	3.86	3.82	6.89	6.98
% Saving	12.30%	13.21%	12.12%	10.92%

Table 6.10: Performance of all one-variable output *cyclic* functions v/s original functions.

Radix	Heur	Number of output *cyclic* functions			Percentage
		Better	Equal	Worse	Success
3	DM	8	14	2	91.67%
	PA	8	14	2	91.67%
4	DM	112	117	23	90.87%
	PA	119	107	26	88.68%
5	DM	1698	1198	224	92.82%
	PA	1785	1092	243	92.21%
6	DM	27814	16153	2683	94.25%
	PA	29275	14388	2987	93.60%

Table 6.11: Performance of all one-variable *increasing* output cyclic function v/s original functions.

Radix	Heur	Number of output *cyclic* functions			Percentage
		Better	Equal	Worse	Success
3	DM	2	5	0	100.00%
	PA	2	5	0	100.00%
4	DM	12	19	0	100.00%
	PA	12	19	0	100.00%
5	DM	52	69	0	100.00%
	PA	51	69	1	99.17%
6	DM	205	251	0	100.00%
	PA	212	240	4	99.12%

Table 6.12: Performance of all one-variable *decreasing* output cyclic functions v/s original functions.

Radix	Heur	Number of output *cyclic* functions			Percentage success
		Better	Equal	Worse	
3	DM	2	5	0	100.00%
	PA	2	5	0	100.00%
4	DM	12	19	0	100.00%
	PA	12	19	0	100.00%
5	DM	52	69	0	100.00%
	PA	54	67	0	99.17%
6	DM	205	251	0	100.00%
	PA	213	237	6	98.68%

Table 6.13: Performance of all one-variable *mixed* output cyclic functions v/s given functions.

Radix	Heur	Number of output *cyclic* functions			Percentage success
		Better	Equal	Worse	
3	DM	4	4	2	80.00%
	PA	4	4	2	80.00%
4	DM	95	72	23	87.89%
	PA	97	70	23	87.89%
5	DM	1594	1060	224	92.20%
	PA	1675	951	252	91.24%
6	DM	27404	15651	2683	94.13%
	PA	28888	13825	3025	93.39%

Table 6.14: Performance of the randomly generated two-variable output cyclic functions v/s original functions.

Radix	Heur	Number of output *cyclic* functions			Percentage Success
		Better	Equal	Worse	
3	DM	1303	1230	467	84.43%
	PA	1285	1231	484	83.87%
4	DM	1656	958	386	87.13%
	PA	1732	870	396	86.73%
5	DM	1984	708	308	89.73%
	PA	2110	533	357	88.10%
6	DM	2202	767	31	98.96%
	PA	2286	678	36	98.80%

Table 6.15: Performance of randomly generated two-variable *increasing* output cyclic functions v/s original functions.

Radix	Heur	Number of output *cyclic* functions			Percentage Success
		Better	Equal	Worse	
3	DM	2175	795	30	99.0%
	PA	2162	802	36	98.8%
4	DM	2563	419	18	99.4%
	PA	2525	460	15	99.5%
5	DM	2612	388	0	100.0%
	PA	2543	454	3	99.9%
6	DM	2654	346	0	100.0%
	PA	2610	390	0	100.0%

Table 6.16: Performance of randomly generated two-variable *decreasing* output cyclic functions v/s original functions.

Radix	Heur	Number of output *cyclic* functions			Percentage
		Better	Equal	Worse	Success
3	DM	246	1236	1518	49.40%
	PA	267	1273	1460	51.33%
4	DM	69	727	2204	26.53%
	PA	114	813	2073	30.90%
5	DM	194	1038	1768	41.07%
	PA	348	1100	1552	48.27%
6	DM	217	1112	1671	44.30%
	PA	389	1210	1599	53.29%

Table 6.17: Performance of randomly generated two-variable *symmetric* output cyclic functions v/s original functions.

Radix	Heur	Number of output *cyclic* functions			Percentage
		Better	Equal	Worse	Success
3	DM	350	244	132	81.82%
	PA	353	234	139	80.85%
4	DM	1783	727	390	87.00%
	PA	1827	813	450	85.00%

Chapter 7

MVL Programmable Logic Array (PLA)

Programmable Logic Arrays (PLAs) are important sub-systems in the design of digital systems. This is due to its regular structure, ease of testability and automatic synthesis. PLAs can also be used to implement complex MVL circuits [26]. In the past, several techniques for optimizing the structure of a PLA have been proposed.

In this chapter we will review some of the existing architectures and optimization techniques for MVL-PLA.

7.1 Existing MVL PLA Realization

In [26] Sasao has proposed three types of MVL-PLA.

1. PLA with MIN and MAX arrays (Type 1).

2. PLA with AND-OR arrays followed by output-encoding (Type 2).

3. PLA with two-bit decoder and permutation network (Type 3).

The sum-of-products expression has been minimized by using various existing PLA tools such as MINI, MINI-II, and ESPRESSO-MV [26]. The three mentioned PLA types are discussed below.

7.1.1 Type 1 PLA

For type 1 PLA, an arbitrary r-valued logic function is represented by an expression:

$$f = 1 \cdot g_1 \vee 2 \cdot g_2 ..... \vee (r-2).g_{r-2} \vee g_{r-1} \tag{7.1}$$

Each subfunction g_i can be represented as:

$$g_i = \bigvee X_1^{s_1} \cdot X_2^{s_2} .....X_n^{s_n} \tag{7.2}$$

where $X_n^{s_n}$ is an *universal literal.* An *universal literal* is a one variable r-valued input two-valued output function. An *universal literal* $X_i^{S_j}$ takes a value of 0 if $X \notin S_j$ and a value $(r-1)$ if $X \in S_j$, where $S_j \in R$ and $R = \{0, 1, 2, \cdots, r-1\}$.

Figure 7.1 shows Type 1 r-valued PLA. It consists of n-input lines, m-output lines, and a MIN-MAX array [26]. It has n *literal generators* each converts r-valued variable x_i into two valued literals where $r \in \{0, 1, 2, ..r-1\}$. For 4-valued logic the literal generator generates four literals $X^{\{1,2,3\}}$, $X^{\{0,2,3\}}$, $X^{\{0,1,3\}}$, $X^{\{0,1,2\}}$. Other

literals can be obtained by logical product of some of the above four literals. Since a literal generator has r outputs and since $(r-2)$ different constants are used in the MIN array, the number of rows in the MIN array is $H_1 = (nr + r - 2)$. The number of columns is equal to the number of product terms in equation 7.1 which is $(r-1)$. Also from equation 7.2, it can be observed that each subfunction g_i can be realized by at most r^{n-1} products. Therefore, the total number of columns 'W' required for realizing an n-variable function is $W \leq (r-1)r^{n-1}$. Since there are m outputs, the number of rows in MAX array is $H_2 = m$. The overall size of type 1 PLA is $S_{type1} = W \times (H_1 + H_2)$.

Type 1 PLA is easily implemented in bipolar technology [26]. Due to the requirement of large number of transistors, type 1 PLA may not be suitable for MOS implementation [26].

Example

Let us design a four valued adder shown in Table 7.1.

We obtain the minimum sum-of-products expressions for sum and carry functions as follows:

$Sum = 1 \cdot g_1 \vee 2 \cdot g_2 \vee g_3$, where

$g_1 = X_1^{\{1,3\}} \cdot X_2^{\{0,2\}} \vee X_1^{\{0,2\}} \cdot X_2^{\{1,3\}}$

$g_2 = X_1^{\{2\}} \cdot X_2^{\{0\}} \vee X_1^{\{1\}} \cdot X_2^{\{1\}} \vee X_1^{\{0\}} \cdot X_2^{\{2\}} \vee X_1^{\{3\}} \cdot X_2^{\{3\}}$

$g_3 = X_1^{\{3\}} \cdot X_2^{\{0\}} \vee X_1^{\{2\}} \cdot X_2^{\{1\}} \vee X_1^{\{1\}} \cdot X_2^{\{2\}} \vee X_1^{\{0\}} \cdot X_2^{\{3\}}$

$Carry = 1 \cdot h_1$, where

Table 7.1: Truth table for four-valued adder.

input		output	
x_1	x_2	*sum*	*carry*
0	0	0	0
0	1	1	0
0	2	2	0
0	3	3	0
1	0	1	0
1	1	2	0
1	2	3	0
1	3	0	1
2	0	2	0
2	1	3	0
2	2	0	1
2	3	1	1
3	0	3	0
3	1	0	1
3	2	1	1
3	3	2	1

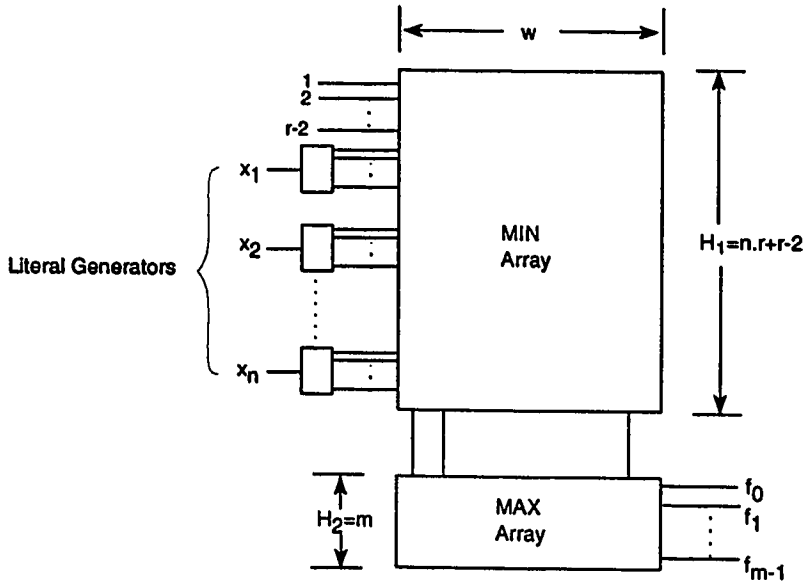


Figure 7.1: r-valued PLA with MIN and MAX array (Type 1 PLA).

$h_1 = X_1^{\{3\}} \cdot X_2^{\{1\}} \vee X_1^{\{2,3\}} \cdot X_2^{\{2\}} \vee X_1^{\{1,2,3\}} \cdot X_2^{\{3\}}$.

Note that 10 product terms are required for sum and 3 product terms are required for carry operation, therefore, 13 product terms are used in this PLA.

7.1.2 Type 2 PLA

This PLA type consists of n-input m-output with AND-OR arrays followed by encoders at the output as shown in Figure 7.2 [26]. In this PLA, external input and output lines take r different values where $r \in \{0, 1, 2, ..r-1\}$. However, the body of the PLA (AND and OR array) is two-valued and manipulates only 0 and $r-1$. The *literal generator* converts an r-valued signal into mp two-valued functions $h_0, h_1, ...h_{mp-1}$, where $p = \lfloor \log_2 r \rfloor$. The output encoder converts two-valued

signals $(h_0, h_1, ..., h_{mp-1})$, into r-valued signals.

Since each *literal generator* has r outputs, therefore the height of the AND array is $H_1 = n.r$. Moreover, since there are m-output and each row of the OR array realizes the function h_j where $0 \leq j \leq p-1$, therefore, the number of rows in the OR array is $H_2 = m.p$. The number of columns 'W' required for r-valued function is $W \leq \lfloor \log_2 r \rfloor \cdot r^{n-1}$. The type 2 PLA is suitable for MOS/CMOS implementation [26].

Example

Let us design the four-valued adder as shown in Table 7.1. Suppose that four-valued output signal is represented by a pair of two-valued signals as shown in Table 7.2. Then the function to be realized by the arrays can be represented as Table 7.3. We obtain the minimum sum-of-products expression for s_1, s_0, and c_0 as follows :

$$s_1 = X_1^{\{2,3\}} \cdot X_2^{\{0\}} \vee X_1^{\{1,2\}} \cdot X_2^{\{1\}} \vee X_1^{\{0,1\}} \cdot X_2^{\{2\}} \vee X_1^{\{0,3\}} \cdot X_2^{\{3\}}$$

$$s_0 = X_1^{\{1,3\}} \cdot X_2^{\{0,2\}} \vee X_1^{\{0,2\}} \cdot X_2^{\{1,3\}}$$

$$c_0 = X_1^{\{3\}} \cdot X_2^{\{1\}} \vee X_1^{\{2,3\}} \cdot X_2^{\{2\}} \vee X_1^{\{1,2,3\}} \cdot X_2^{\{3\}}$$

Note that 9 product terms are required for realizing the adder.

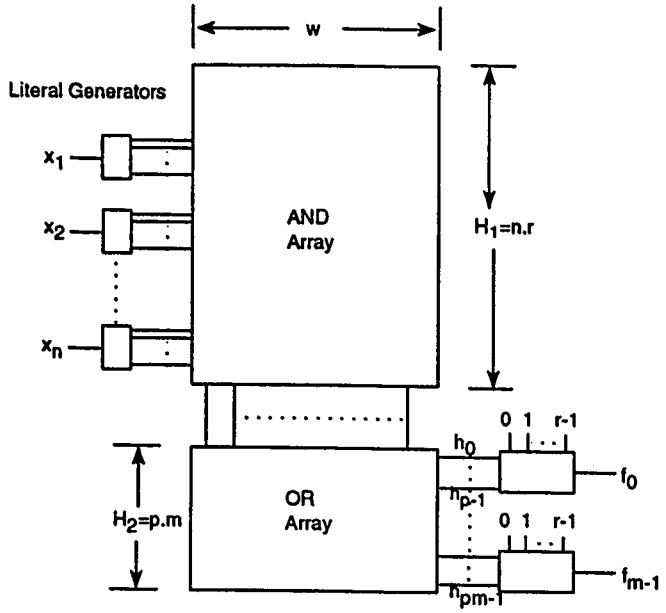


Figure 7.2: r-valued PLA with output encoder (Type 2 PLA).

Table 7.2: Output encoding for four-valued adder.

4-valued signal	2-valued signals
0	0 0
1	0 3
2	3 0
3	3 3

Table 7.3: Truth table of adder for Type 2 PLA.

Input		Output			
		sum		carry	
x_1	x_2	s_1	s_0	c_1	c_0
0	0	0	0	0	0
0	1	0	3	0	0
0	2	3	0	0	0
0	3	3	3	0	0
1	0	0	3	0	0
1	1	3	0	0	0
1	2	3	3	0	0
1	3	0	0	0	3
2	0	3	0	0	0
2	1	3	3	0	0
2	2	0	0	0	3
2	3	0	3	0	3
3	0	3	3	0	0
3	1	0	0	0	3
3	2	0	3	0	3
3	3	3	0	0	3

7.1.3 Type 3 PLA

Type 3 PLA has almost the same structure as that of Type 2 PLA. In addition, it has a two-bit decoder and a permutation network at the input (Figure 7.3) [26]. The size of this type of PLA can be reduced by specific assignment of input variables to two-bit decoders. The *literal generator* generates two literals $X^{\{2,3\}}$, $X^{\{1,3\}}$ as opposed to 4 literals used in Type 1 and Type 2 PLA. The permutation network has been used to group outputs from 2-bit decoders in such a way as to minimize the size of the PLA.

Example

Let us design the adder of four-valued logic as shown in Table 7.1. Suppose four independent two-valued variables y_1, y_2, y_3, and y_4 represent X_1 and X_2.

$y_1 = X_1^{\{2,3\}}, y_2 = X_1^{\{1,3\}}, y_3 = X_2^{\{2,3\}}, y_4 = X_2^{\{1,3\}}$ Then

$\overline{y}_1 = X_1^{\{0,1\}}, \overline{y}_2 = X_1^{\{0,2\}}, \overline{y}_3 = X_2^{\{0,1\}}, \overline{y}_4 = X_2^{\{0,2\}}$

By using the new variables y_1, y_2, y_3, and y_4 the adder can be represented as shown in Table 7.4. Next introduce two super variables $Y_1 = (y_1, y_3)$ and $Y_2 = (y_2, y_4)$. Then the minimum sum-of-product expression for S_1, S_0 and C_0 is obtained as .

$S_1 = Y_1^{\{03,30\}} \cdot Y_2^{\{00,03,30\}} \vee Y_1^{\{00,33\}} \cdot Y_2^{\{33\}}$

$S_0 = Y_2^{\{03,30\}}$

$C_0 = Y_1^{\{33\}} \vee Y_1^{\{03,30\}} \cdot Y_2^{\{33\}}$

As can be seen, the PLA realizing these functions require only five products.

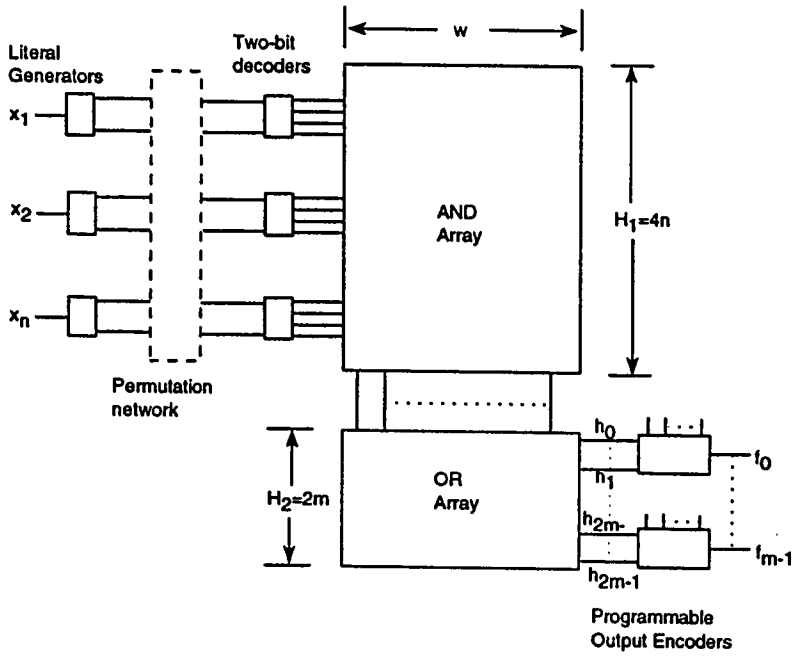


Figure 7.3: r-valued PLA with 2-bit decoders and programmable output encoders (Type 3 PLA).

Table 7.4: Truth table of adder for Type 3 PLA.

X_1		X_2		sum		carry	
y_1	y_2	y_3	y_4	s_1	s_0	c_1	c_0
0	0	0	0	0	0	0	0
0	0	0	3	0	3	0	0
0	0	3	0	3	0	0	0
0	0	3	3	3	3	0	0
0	3	0	0	0	3	0	0
0	3	0	3	3	0	0	0
0	3	3	0	3	3	0	0
0	3	3	3	0	0	0	3
3	0	0	0	3	0	0	0
3	0	0	3	3	3	0	0
3	0	3	0	0	0	0	3
3	0	3	3	0	3	0	3
3	3	0	0	3	3	0	0
3	3	0	3	0	0	0	3
3	3	3	0	0	3	0	3
3	3	3	3	3	0	0	3

K. Vijayan Asari and C. Eswaran [50] proposed an optimization technique to reduce the size of the Type 1 and Type 2 PLA proposed by Sasao [26]. Both types of PLA include multiple function literal circuits (MFLC) for the purpose of minimization. The inputs of MFLC $x_1, x_2, ..., x_n$ are r-valued functions. The output $(x_1, x_2, ..., x_n)^S$ takes on a value of $r-1$ if $(x_1, x_2, ..., x_n) \in S \subseteq R$, otherwise it is 0, where $R = \{(00...0), (00...1), (r-1, r-1, ..., r-1)\}$. There are $(2^r)^n$ possible output terms for the MFLC. The MFLC which generates the basic terms is called as MFLCB (multiple function literal circuit block). Basic terms are set of terms from which other terms can be generated. Figure 7.4 shows the modified form of Sasao's Type 1 MVPLA which uses MFLCB's at the input. Each MFLCB has p inputs, where $1 < p < n$, and there are n/p MFLCB's. It has been shown that the modified PLA has the following height, width and size, respectively

$$H'_{1Type1} = \frac{n}{p} \cdot r^p + (r-2)$$

$$W'_{Type1} \leq (r-1) \cdot r^{(n-p)}$$

$$S'_{Type1} = (H'_{1Type1} + H'_{2Type1}) \cdot W'_{Type1}$$

The above equations show that $H'_1 > H_1$, $W' < W$, and $S' < S$, where H_1, W and S are the height, width and overall size of the MVL-PLA proposed by Sasao [26]. It is evident from above equations that the overall size of the proposed PLA is much smaller than the size of the PLA proposed by Sasao [26].

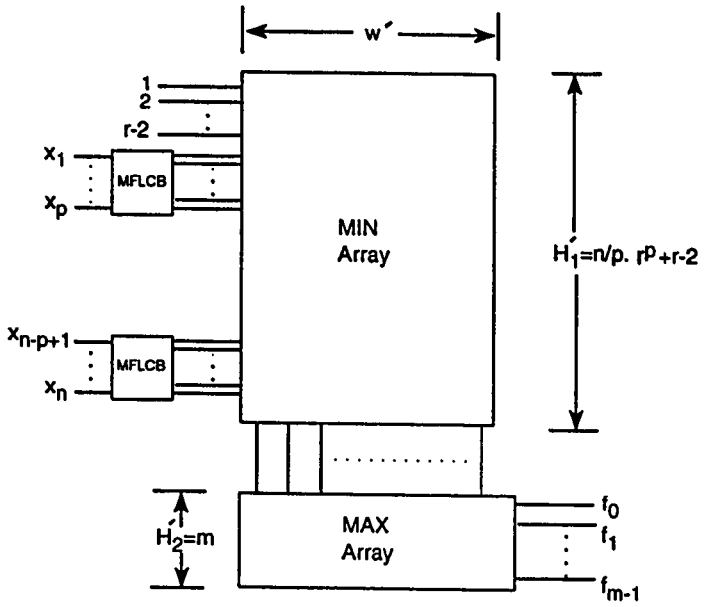


Figure 7.4: Modified form of Sasao's Type 1 PLA.

The modified Type 2 PLA is almost the same as that of Sasao's Type 2 PLA with the only exception that the modified MVL-PLA uses MFLCBs at the input instead of *literal generators* as shown in Figure 7.5. The r-valued signals are converted into two-valued multiple functions literals by MFLCB's. These two-valued multiple signals are converted into r-valued output signals $f_0, f_1, ..., f_{m-1}$ by the output encoders. The heights of the AND array and OR array are given by

$$H'_{1Type2} = \frac{n}{p} \cdot r^p$$

$$H'_{2Type2} = m \cdot l$$

where $l = log_2 P$

The width of the OR array is given as follows

$$W'_{Type2} \leq [log_2 r] \cdot r^{(n-p)}$$

Therefore, the overall size of the PLA is

$$S'_{Type2} = (H'_{1Type2} + H'_{2Type2}) \cdot W'_{Type2}$$

It can be observed from the above equations that the size of the proposed PLA is smaller than the size of the MVL-PLA proposed by Sasao [26].

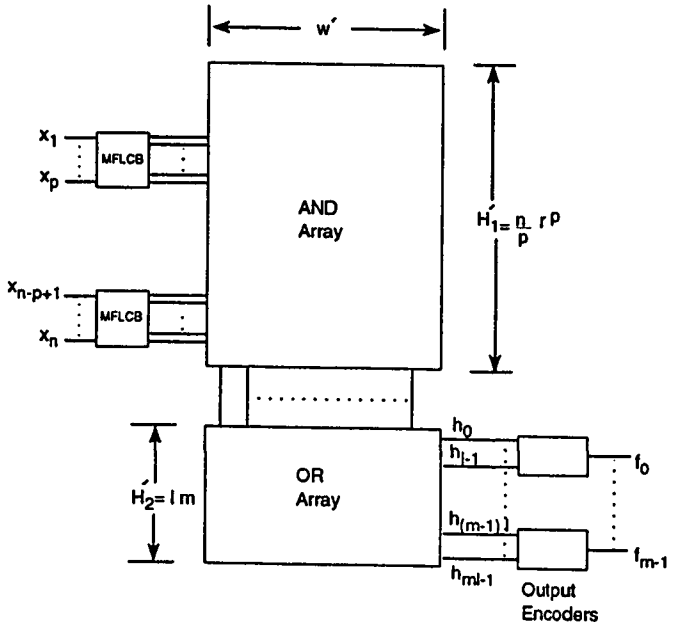


Figure 7.5: Modified form of Sasao's Type 2 PLA.

Chapter 8

New MVL-PLA Structures

Several MVL-PLA structures has been mentioned in Section 7.1. In this chapter we will discuss some new MVL-PLA structures for the implementation of multi-valued logic functions. The Current-Mode CMOS technology will be used to realize the MVL-PLA structures [8]. As the name implies, current is used for information transfer and storage in this technology. The measure of current represents logic values and the direction of current flow determines the sign of the logic value.

The MVL-PLA structures realization uses four Current-Mode CMOS building blocks namely, *cycle, min, tsum* and *constants.* Previously, no MVL-PLA architectures have been realized using these building blocks. Definitions of these building blocks has been discussed in Section 2.1.

The new MVL-PLA structures are

1. PLA Type 'A'

2. PLA Type 'B'

3. PLA Type 'C'

8.1 PLA Type 'A'

Figure 8.1 shows an n-input and m-output r-valued PLA with MIN and TSUM array. It has n *cyclic generators*. Each *cyclic generator* generates r cyclic terms. A given MVL function is realized as the TSUM of MIN of the cyclic functions. It is analogous to the sum of product form in the binary system. The *cyclic generators*, MIN and TSUM array can be implemented in Current Mode CMOS logic. Each *cyclic generator* generates a number of literal lines equal to the radix of the system, r. The constant lines are used to realize the constant functions, for example, for 4-valued logic the constant functions $\langle 0000 \rangle$, $\langle 1111 \rangle$, $\langle 2222 \rangle$, and $\langle 3333 \rangle$ are realized by the constant lines. Since there are r constant lines and each *cyclic generator* generates r cyclic functions, therefore, the number of rows in the MIN array is $H_1 = nr + r$. Similary, the number of rows in the TSUM array is equal to the number of output lines i.e. $H_2 = m$. The functions $\langle 0001 \rangle, \langle 0010 \rangle, \langle 0100 \rangle, \langle 1000 \rangle$ constitute the basic functions for 4-valued logic from which other functions can be

realized by performing *tsum* operations on them. All these functions require one column for realization. Since the worst case function for 4-valued logic $\langle 3332 \rangle$ require 11, i.e. $(r(r-1))$ columns for realization, therefore, the total number of columns 'W' required for realizing an n-variable function is $W \leq (n \times \{r(r-1)-1\})$. Therefore, the upper bound on the area of type 'A' PLA is $\leq W \times (H_1 + H_2)$.

Example

As an example let us consider the realization of the function $\langle 3021 \rangle$ in a 4-valued system, given in Figure 8.2. The function is realized by taking the *tsum* of *min* of the cyclic functions.

$$(x^{\rightarrow 0} \bullet x^{\rightarrow 1} \bullet x^{\rightarrow 3}) \uplus (x^{\rightarrow 2} \bullet x^{\rightarrow 3}) \uplus (x^{\rightarrow 1} \bullet x^{\rightarrow 3}) = 0010 \uplus 2001 \uplus 1010 = 3021$$

From the above equation we can observe that 3 columns are required to realize the function $\langle 3021 \rangle$. The realization of the function $\langle 121 \rangle$ in 3-valued system is given as

$$(x^{\rightarrow 1}) \uplus (x^{\rightarrow 0} \bullet x^{\rightarrow 2}) = 120 \uplus 001 = 121$$

Similarly the realization of $\langle 42301 \rangle$ in a 5-valued system is obtained as

$$(x^{\rightarrow 1} \bullet x^{\rightarrow 2}) \uplus (x^{\rightarrow 2} \bullet x^{\rightarrow 3} \bullet x^{\rightarrow 4}) \uplus (x^{\rightarrow 1} \bullet x^{\rightarrow 2} \bullet x^{\rightarrow 3} \bullet x^{\rightarrow 4}) = 12300 \uplus 20001 \uplus 10000 = 42301$$

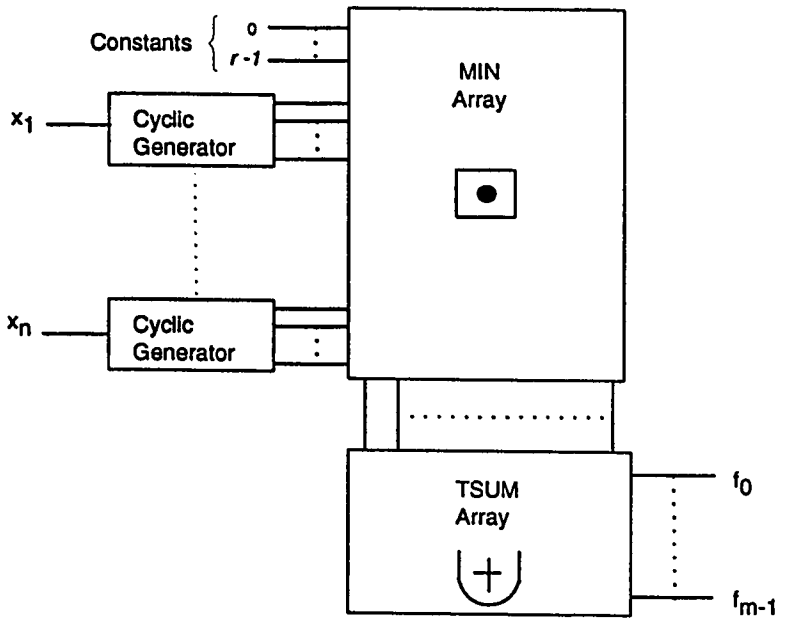


Figure 8.1: r-valued PLA with MIN and TSUM array (Type 'A').

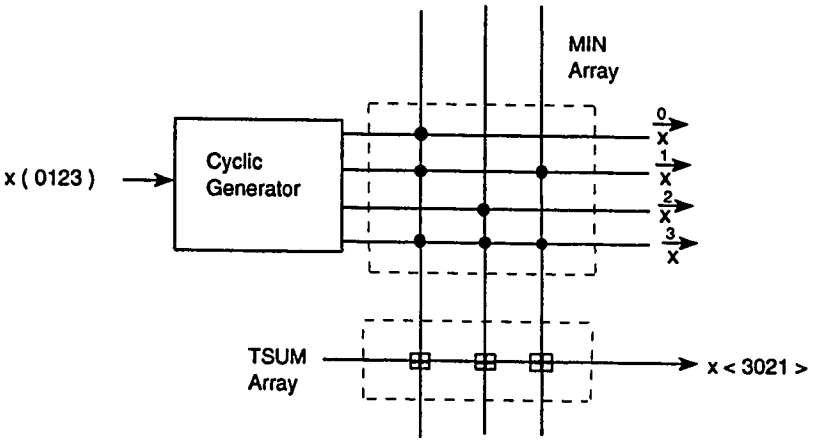


Figure 8.2: Realization of $\langle 3021 \rangle$ for 4-valued system with Type 'A' PLA.

8.2 PLA Type 'B'

The architecture of Type 'B' PLA is similar to Type 'A', but here the first level uses a TSUM array and the second level uses a MIN array. There are n *cyclic generators* and each generates r cyclic functions. The architecture of Type 'B' PLA is shown in Figure 8.3. It is analogous to the product of sum in binary system. As an example, for 4-valued system the realization of the function $\langle 3211 \rangle$ using type 'B' PLA is shown in Figure 8.4. The function is realized by taking the *min* of *tsum* of the cyclic functions.

$$(x^{\rightarrow 2} \uplus x^{\rightarrow 3}) \bullet (x^{\rightarrow 1} \uplus x^{\rightarrow 3}) \bullet (x^{\rightarrow 1} \uplus x^{\rightarrow 2}) = 3313 \bullet 3232 \bullet 3331 = 3211$$

From the above equation we can observe that the realization of the function $\langle 3211 \rangle$ requires 3 columns. The realization of the function (110) for 3-valued system is given as

$$(x^{\rightarrow 1}) \bullet (x^{\rightarrow 0} \uplus x^{\rightarrow 2}) = 120 \bullet 212 = 110$$

Similarly, the realization of function $\langle 12343 \rangle$ in 5-valued system is obtained as

$$(x^{\rightarrow 0} \uplus x^{\rightarrow 1}) \bullet (x^{\rightarrow 1} \uplus x^{\rightarrow 4}) \bullet (x^{\rightarrow 0} \uplus x^{\rightarrow 3} \uplus x^{\rightarrow 4}) = 13444 \bullet 42443 \bullet 44344 = 12343$$

The disadvantage of this architecture is that it realizes only a small number of functions, whereas other functions can not be realized even by using a large number of columns. For example, in 3-valued logic the function $\langle 002 \rangle$ can not be realized by performing *min* operation with any other function. This lead to the development of

another PLA architecture which provides more functionality by using extra *cyclic* gates at the input. This PLA is referred to as type 'C' PLA.

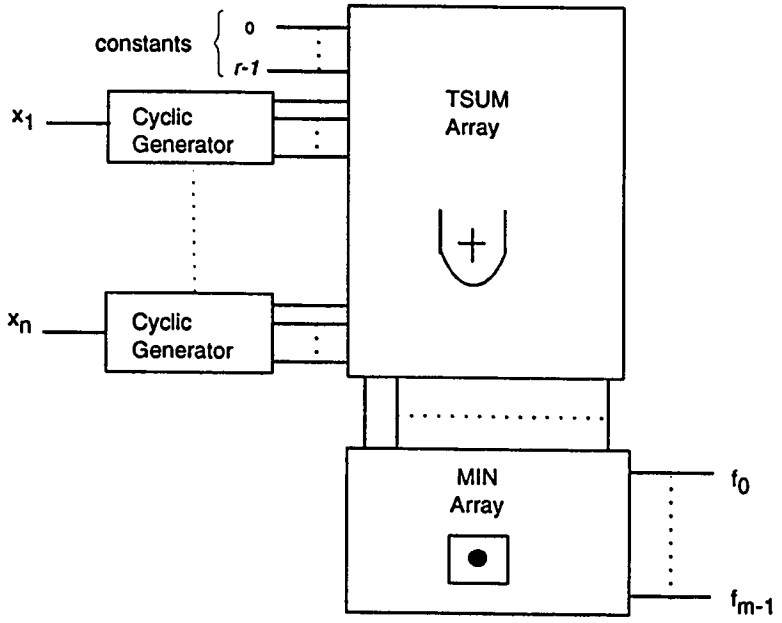


Figure 8.3: r-valued PLA with TSUM and MIN array (Type 'B').

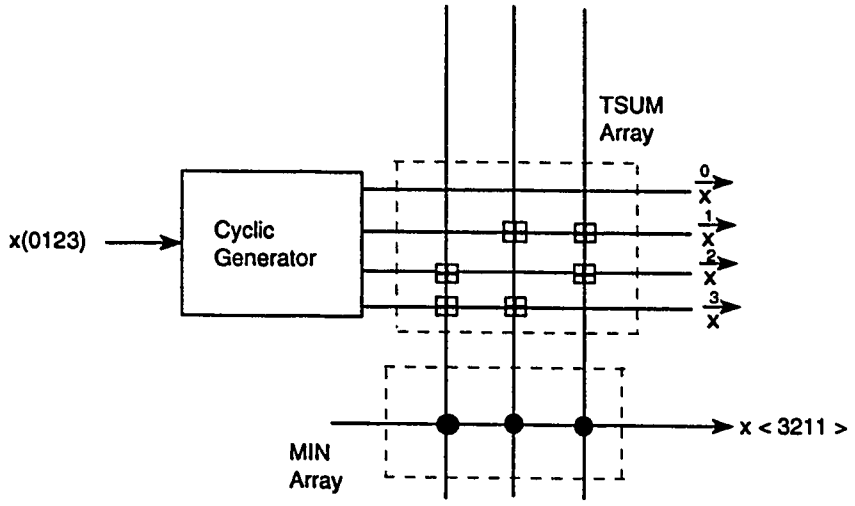


Figure 8.4: Realization of ⟨3211⟩ for 4-valued system with Type 'B' PLA.

8.3 PLA Type 'C'

The architecture of Type 'C' PLA is similar to Type 'B' PLA, but here we are introducing extra *cycle* gates in the *cyclic generators* block to provide more functional coverage. In this architecture, the cyclic generators consists of both *clockwise* and *counter-clockwise* cyclic operators. There can be at most $(r-1)$ extra cyclic generators and each cyclic generator can generate r cyclic functions. Moreover, there are r constant lines to realize the constant functions. Therefore, the number of rows in the TSUM array for n-variable function is $H_1 = r + n \times (2r-1)$. The number of rows in the MIN array is equal to the number of output lines i.e. $H_2 = m$. Since, by performing the *min* operation $(x^{\rightarrow^0} \bullet x^{\rightarrow^1} \bullet \cdots x^{\rightarrow^{r-1}})$ produces $(00\cdots0)$, therefore by adding extra columns no more additional functions can be obtained. From this

observation, it is concluded that the number of columns 'W' required for n-variable functions can be at most nr. Therefore, the upper bound on the area of type 'C' PLA is $\leq W \times (H_1 + H_2)$.

In type 'C' PLA, the *tsum* operation is followed by *min* operation. So when the number of cyclic gates are increased, it is possible to perform *tsum* operation between identical cyclic functions (e.g $x^{\rightarrow 2} \uplus x^{\rightarrow 2}$) without utilizing extra columns. The identical functions may be realized either through *clockwise* or *counter-clockwise* cyclic functions. For example, for 3-valued system, the *clockwise* cyclic function $x^{\rightarrow 2}$ and *counter-clockwise* cyclic function $x^{\rightarrow 1}$ are similar. Thus the advantage of using extra cycle gate is that it provides more functional coverage as compared to Type 'A' and Type 'B' PLA using smaller number of columns. For example, type 'A' PLA and type 'B' PLA realizes the function $\langle 102 \rangle$ in 3-valued logic using more than two columns. However, the same function can be realized by type 'C' PLA using only two columns as shown in Figure 8.5. The function is realized by taking the *min* of *tsum* of the cyclic functions. It is analogous to the product of sum in binary system.

$$(x^{\rightarrow 1} \uplus x^{\rightarrow 2}) \bullet (x^{\rightarrow 0} \uplus x^{\rightarrow 1}) = 202 \bullet 122 = 102$$

Similarly for 4-valued logic, the function $\langle 3102 \rangle$ requires only two columns for realization using type 'C' PLA. However, type 'A' PLA and type 'B' PLA require more than two columns to realize the same function. The realization of the function

$\langle 3102 \rangle$ using type 'C' PLA is given as

$$(x^{\leftarrow 2} \uplus x^{\rightarrow 2}) \bullet (x^{\rightarrow 0} \uplus x^{\rightarrow 3}) = 3302 \bullet 3133 = 3102$$

Another PLA structure is possible if we introduce extra cycle gates in type 'A' PLA. But since in type 'A' PLA, the *min* operation is followed by *tsum* operation. Therefore, when we perform the ***min*** operation between similar cyclic functions, it will give the same result as the original function. Therefore, this approach is not suitable as it provides the same functional coverage provided by type 'A' PLA at a higher cost.

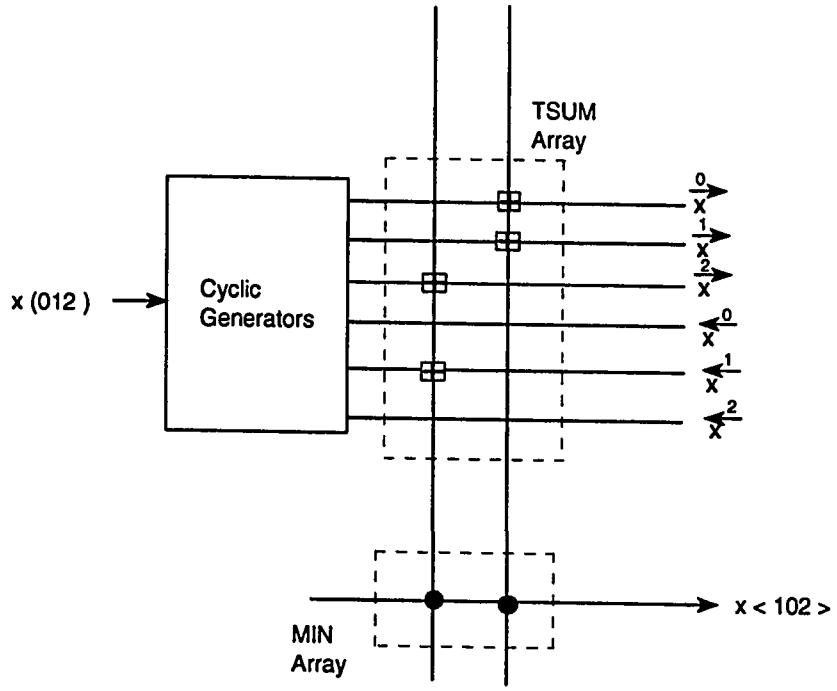


Figure 8.5: Realization of $\langle 102 \rangle$ for 3-valued system with Type 'C' PLA.

8.4 Comparison of the PLA structures

Since again MVL function can be realized by using either PLA type 'A' or PLA type 'B', we will try to assess the merits of the above PLA configurations by enumerating the number of MVL functions produced by each. As a basis for the comparison we will consider the set of all 4-valued one-variable functions (there are 256 such functions). We will determine how many functions are realizable by the above PLAs for various number of columns, c. A function may be realized by different number of columns. We will select the minimum number of columns required to realize a given function.

A program has been written to enumerate the MVL functions according to the number of columns required for their realization. Starting with MVL functions realized by a single column, all functions realized with two columns were generated, then with three columns, etc. From these functions only those functions are selected which require smaller number of columns.

For 4-valued logic the number of functions realized using one column of type 'A' PLA is 14 (i.e 5.47% of the total) and the number of additional functions realized using two columns is 86 (i.e 33.60 % of the total). There are additional 128 functions (i.e 50.00 % of the total) which can be realized by three columns. Functions $\langle 0000 \rangle$, $\langle 1111 \rangle$, $\langle 2222 \rangle$ and $\langle 3333 \rangle$ are realized by constants. Thus the number of functions realized using constants, one, two and three columns for type 'A' PLA are 232 (i.e

90.62 % of the total). Table 8.5 show 24 functions which require more than three columns to realize them.

Similarly by using one column the number of functions realized by type 'B' PLA is 11 (i.e. 4.30% of the total). It realizes only 32 functions (i.e. 12.5% of the total) and 34 functions (i.e. 13.28% of the total) using two and three columns respectively. Functions ⟨0000⟩, ⟨1111⟩, ⟨2222⟩ and ⟨3333⟩ are realized by constants. We observe that 175 functions are still not realizable. Therefore, the total number of functions realized by the combinations of constants, one, two and three columns are 81 (i.e. 31.64% of the total). From these results it is observed that type 'A' PLA provides more functionality than type 'B' PLA.

Similarly, for type 'C' PLA, the number of functions realized by using one column is 22 (i.e. 8.6% of the total). By using two columns and three columns, type 'C' PLA can realize 108 (i.e. 42.19% of the total) and 114 (i.e. 44.53% of the total) respectively. The constant line realizes the functions ⟨0000⟩, ⟨1111⟩,⟨2222⟩ and ⟨3333⟩. By using constant line and combination of one, two and three columns 248 functions are realized by type 'C' PLA. Thus it provides a functional coverage of 96.87%. There are 8 more functions which are not realized by combination of constant, one, two, and three columns. These functions require more than three columns for their realization. These functions are shown in Figure 8.6.

From these results it is observed that type 'C' PLA provides more functional coverage as compared to type 'A' and type 'B' PLA. However this is obtained at

the expense of extra cycle gate. Similar analysis has been carried out for radix 3 and radix 5. Also for radix 3 and radix 5, type 'C' PLA provides more functional coverage than type 'A' PLA and type 'B' PLA. It is also observed that functional coverage decreases with the increase in radix.

The results for radix 3, radix 4, and radix 5 are summarized in Table 8.1, Table 8.2, and Table 8.3 respectively. Table 8.4 show the functions which are not realized by combination of constant, one, two and three columns. These functions need more than three columns to realize them.

Comparison of the PLA structures using *modsum* as combining operator

Another analysis is carried out by using the *modsum* as combining operator instead of the *tsum* operator. For example, for type 'A' PLA, a function is realized by taking the *modsum* of *min* of the *cyclic* functions. Similarly, in type 'B' PLA, a function is represented by *min* of *modsum* of the *cyclic* functions. Type 'C' PLA is represented in the same way as type 'B' PLA, but it requires extra cyclic gate.

Similar analysis has been carried out as done in Section 8.4. The results for radix 3, radix 4 nad radix 5 has been depicted in Table 8.7, Table 8.8 and Table 8.9 respectively. From these tables, it can be observed that type 'B' and type 'C' PLA provide almost the same functional coverage. Therefore, there is no much advantage of using extra cycle gates for type 'C' PLA. However, type 'A' PLA gives better results than type 'B' and type 'C' PLA in terms of the functional coverage.

Type 'A' PLA provides a functional coverage of 100% and 92.96%, 68.45 % for radix 3, radix 4 and radix 5 respectively.

Table 8.1: No. of functions realized using *tsum* as combining operator (R=3).

No. of columns	Number of Realizable Functions		
	Type 'A' PLA	Type 'B' PLA	Type 'C' PLA
1	6	7	9
2	15	8	15
3	3	0	0
Constant	3	3	3
Total Functions	27	18	27
Functional Coverage	100%	66.66%	100%

Table 8.2: No. of functions realized using *tsum* as combining operator (R =4).

No. of columns	Number of Realizable Functions		
	Type 'A' PLA	Type 'B' PLA	Type 'C' PLA
1	14	11	22
2	86	32	108
3	128	34	114
Constant	4	4	4
Total functions	232	81	248
Functional Coverage	90.625%	31.64%	96.875%

Table 8.3: No. of functions realized using *tsum* as combining operator (R =5).

No. of columns	Number of Realizable Functions		
	Type 'A' PLA	Type 'B' PLA	Type 'C' PLA
1	29	20	45
2	363	120	433
3	1407	265	1590
Constant	5	5	5
Total functions	1804	410	2073
Functional Coverage	57.77%	13.12%	66.34 %

Table 8.4: Functions not realized using *tsum* as combining operator for type 'B' PLA using combination of constant, one, two and three columns.

Functions not realized by type 'B' PLA for R=3		
⟨002⟩	⟨020⟩	⟨021⟩
⟨022⟩	⟨102⟩	⟨200⟩
⟨202⟩	⟨210⟩	⟨220⟩

Table 8.5: Functions not realized using *tsum* as combining operator for type 'A' PLA using combination of constant, one, two and three columns.

Functions not realized by type 'A' PLA for R=4		
⟨0031⟩	⟨0310⟩	⟨0311⟩
⟨0312⟩	⟨0313⟩	⟨0332⟩
⟨1003⟩	⟨1031⟩	⟨1103⟩
⟨1203⟩	⟨1303⟩	⟨1322⟩
⟨2031⟩	⟨2033⟩	⟨2132⟩
⟨2213⟩	⟨3031⟩	⟨3100⟩
⟨3110⟩	⟨3120⟩	⟨3130⟩
⟨3203⟩	⟨3221⟩	⟨3320⟩

Table 8.6: Functions not realized using *tsum* as combining operator for type 'C' PLA using combination of constant, one, two and three columns.

Functions not realized by type 'C' PLA for R=4		
⟨1112⟩	⟨1121⟩	⟨1122⟩
⟨1211⟩	⟨1221⟩	⟨2111⟩
⟨2112⟩	⟨2211⟩	

Table 8.7: No. of functions realized using *modsum* as combining operator (R=3).

No. of columns	Number of Realizable Functions		
	Type 'A' PLA	Type 'B' PLA	Type 'C' PLA
1	6	7	8
2	15	9	9
3	3	0	0
Constant	3	3	3
Total functions	27	19	20
Functional Coverage	100%	70.37%	74.07%

Table 8.8: No. of functions realized using *modsum* as combining operator (R =4).

No. of columns	Number of Realizable Functions		
	Type 'A' PLA	Type 'B' PLA	Type 'C' PLA
1	14	12	12
2	82	42	43
3	140	8	8
Constant	4	4	4
Total functions	240	66	67
Functional Coverage	92.96%	25.78%	26.17%

Table 8.9: No. of functions realized using *modsum* as combining operator (R=5).

No. of columns	Number of Realizable Functions		
	Type 'A' PLA	Type 'B' PLA	Type 'C' PLA
1	30	20	25
2	358	190	238
3	1746	170	186
Constant	5	5	5
Total functions	2139	385	454
Functional Coverage	68.45%	12.32%	14.52%

Chapter 9

Conclusions and Future Work

9.1 Conclusions

MVL circuits allow signals to have more than two-levels. MVL allows the possibility of reducing interconnection wiring and increasing the active chip area in integrated circuits. It also has the potential for decreasing the number of pins in an IC package. Reduced noise margins and the lack of an established market for MVL components are some of the drawbacks of the MVL circuits.

In this thesis some of the existing MVL circuit elements and operators have been discussed. All these circuit elements and operators have been designed using current-mode CMOS technology. In this technology current is used to represent multiple-valued logic values. The logic levels are represented as an integer multiple of some reference current ($I_o = 20\mu A$). For example, logic zero is represented by

$I_0 = 0$, logic 1 $= I_0$, logic 2 $= 2I_o$ and so on. The basic elements of current-mode CMOS are: *sum, constant, current-mirror, threshold,* and *switch.* These circuit elements have been used in the design of some MVL operators such as *min, tsum* and *cycle* operators. The functionality of these circuits have been verified by SPICE transient analysis simulation.

The design of logic networks has many constraints such as cost, speed, fan-in, fan-out, etc. The cost constraints led to the development of many simplification techniques. These simplification techniques are aimed at reducing the cost by minimizing the number of basic elements in the network. Some new MVL synthesis techniques for the simplification of circuits have been developed. These are based on the decomposition based mapping technique. The synthesis problem is formulated as a mapping from an input matrix to an output matrix. The *matching count matrix* has been used to minimize the number of switching operations required to realize a given MVL function. The techniques proposed were :

1. Output phase assignment with complement.

2. Input phase assignment without complement.

3. Input phase assignment with complement.

Experiments were carried out to test the effectiveness of the above proposed techniques and the existing output phase without complement technique. The percentage saving in the number of switching elements has been taken as a measure of

performance. All the techniques give a substantial percentage saving in the number of switching elements. It has also been observed that on the average, *input phase assignment with complement* technique gives better percentage saving as compared to the other techniques. Furthermore, on the average *input phase assignment* techniques give more percentage saving than the *output phase assignment* techniques. The *input phase assignment* techniques also do not require any hardware circuitry at the output to get back the original function.

An optimal output phase optimization technique referred to as *output phase cyclic* technique has been developed to reduce the number of implicants for the sum-of-product MVL expression. The DM and PA heuristics were used to obtain the optimized sum-of-product expressions. It has been observed that the proposed technique provides smaller number of implicants for the output *cycled* function as compared to the original function. Moreover, the number of functions which are required to be minimized to choose the optimal function is less as compared to the existing output phase *permuted* functions. Only r number of functions are required to be minimized by our proposed technique, whereas, $r!$ functions are needed to be minimized by the existing output phase *permuted* technique. For example, for 7-valued functions, our proposed technique requires only 7 functions to be minimized to get the optimal solution as compared to 5040 functions which are required by the existing technique. Further, the original functions are obtained by introducing a *counter-clockwise* cyclic operator at the output, whereas no realization of such

unary operation has been proposed by the existing technique.

Some new MVL-PLA structures have also been proposed in this thesis. Two main classes of MVL-PLA were proposed. The first one has used *tsum* and the other one has used *modsum* as the combining operator to represent the sum-of-product expression. Each class of PLA has three types of configurations. Type 'A' PLA configuration, has used MIN as the first level and TSUM as the second level in the structure, whereas, Type 'B' PLA structure has used TSUM in the first level and MIN in the second level. Type 'C' PLA is similar to type 'B' PLA, but it has used extra *cycle* gate at the input to provide more functionality. It has been observed that type 'C' PLA provides more functional coverage as compared to type 'A' and 'B' PLA when *tsum* is used as combining operator. Similarly, for *modsum* as combining operator, type 'A' provides more functional coverage as compared to type 'B' and type 'C' PLA.

9.2 Future Directions

- Generation of standard cell layouts of the MVL circuits to automate the design cycle of MVL functions. These standard cell layouts can be used in the design of various MVL applications.

- Realization of the MVL circuits and operators using technologies such as BiC-MOS and GaAs MESFET.

- Developing some other possibilities for decomposition based mapping technique. One possibility might be to select both the *cyclic* and *complement* of cyclic functions in the output or input matrix. For example, one can pair (x_i, $\vec{y_i}$) and (x_i, $\overrightarrow{\overline{y_i}}$) to get the total maximum matching count.

- In this thesis, experiments have been carried out to find the functional coverage for only unary input PLA. It can be extended for more than one variable.

Appendix A

SPICE Simulation of MVL circuit elements and operators.

SUM.CIR

```
.TRAN .05U 2U
.PRINT TRAN I(V2) I(V8) I(V14)
I2 28 27 PWL (0.4U 0 0.4001U 20U 0.8U 20U 0.8001U 40U 1.2U 40U
+ 1.2001U 60U 1.6U 60U 1.6001U 0 2.0U 0 2.001U 20U 2.4U 20U )
I3 28 1 PWL (0.4U 0 0.4001U 20U 0.8U 20U 0.8001U 20U 1.2U 20U
+ 1.2001U 0U 1.6U 0U 1.6001U 0 2.0U 0 2.001U 20U 2.4U 20U)
V8 27 0
V14 1 0
V2 0 28
.END
```

CONSTANT.CIR

```
M1 1 2 2 1 P W=3U L=6U
M2 2 2 0 0 N W=3U L=3U
M3 3 2 0 0 N W=6U L=3U
VSENSE 4 3 0
VD1 1 0 5
VD2 4 0 5
.MODEL N NMOS (LEVEL=3 VTO=0.7 KP=40E-6 GAMMA=1.1 PHI=0.6
+ PB=0.7 CGSO=3E-10 CGDO=3E-10 CGBO=5E-10 RSH=25 CJ=4.4E-4 MJ=0.5
+ CJSW=4E-10 MJSW=0.3 JS=1E-5 TOX=5E-8 NSUB=1.7E16 TPG=1
```

```
+ XJ=6E-7 LD=3.5E-7 UO=775)
.MODEL P PMOS (LEVEL=3 VTO=-0.8 KP=12E-6 GAMMA=0.6 PHI=0.6
+ PB=0.7 CGSO=2.5E-10 CGDO=2.5E-10 CGBO=5E-10
+ RSH=80 CJ=1.5E-4 MJ=0.6 CJSW=4E-10 MJSW=0.6 JS=1E-5
+ TOX=5E-8 NSUB=5E15 TPG=1 XJ=5E-7 LD=2.5E-7 UO=250)
.OPTION NOPAGE
.TRAN 0.1US 6US
.PRINT TRAN V(2) I(VSENSE)
.END
```

MIRROR.CIR

```
M1 1 1 0 0 N W=3U L=3U
M2 2 1 0 0 N W=6U L=3U
M3 4 1 0 0 N W=9U L=3U
M4 7 5 5 7 P W=3U L=3U
M5 8 6 6 8 P W=3U L=3U
I 0 3 20U
VM1 3 1 0
VM2 5 2 0
VM3 6 4 0
VD1 7 0 5V
VD2 8 0 5V
.MODEL N NMOS (LEVEL=3 VTO=0.7 KP=40E-6 GAMMA=1.1 PHI=0.6
+ PB=0.7 CGSO=3E-10 CGDO=3E-10 CGBO=5E-10 RSH=25 CJ=4.4E-4 MJ=0.5
+ CJSW=4E-10 MJSW=0.3 JS=1E-5 TOX=5E-8 NSUB=1.7E16 TPG=1
+ XJ=6E-7 LD=3.5E-7 UO=775)
.MODEL P PMOS (LEVEL=3 VTO=-0.8 KP=12E-6 GAMMA=0.6 PHI=0.6
+ PB=0.7 CGSO=2.5E-10 CGDO=2.5E-10 CGBO=5E-10
+ RSH=80 CJ=1.5E-4 MJ=0.6 CJSW=4E-10 MJSW=0.6 JS=1E-5
+ TOX=5E-8 NSUB=5E15 TPG=1 XJ=5E-7 LD=2.5E-7 UO=250)
.TRAN 0.1US 6.4US
.PRINT TRAN I(VM1) I(VM2) I(VM3)
.END
```

THRESHOLD.CIR

```
.TRAN 0.1U 6U
.PRINT TRAN I(V3) V(38)
M3 38 8 7 4 PM L=3U W=3U
V1 7 0 5V
V3 25 0
M10 38 11 0 12 NM L=4U W=6U
V4 11 0 1.61V
I1 8 19 PWL 0 0 1U 0 1U 20U 2U 20U 2U 40U 3U 40U
+ 3U 60U 3.99U 60U 3.99U 0
M23 19 25 25 0 NM L=3U W=3U
M2 8 8 7 3 PM L=3U W=3U
.END
```

SWITCH.CIR

```
.TRAN 0.1U 6U
.PRINT TRAN I(V2) I(V3) V(1)
.MODEL NM NMOS (LEVEL=3 VTO=0.7 KP=40E-6 GAMMA=1.1 PHI=0.6
+ PB=0.7 CGSO=3E-10 CGDO=3E-10 CGBO=5E-10 RSH=25 CJ=4.4E-4 MJ=0.5
+ CJSW=4E-10 MJSW=0.3 JS=1E-5 TOX=5E-8 NSUB=1.7E16 TPG=1
+ XJ=6E-7 LD=3.5E-7 UO=775)
V1 1 0 5V
I1 5 0 PWL 0 0 1U 0 1U 20U 2U 20U 2U 40U 3U 40U
+ 3U 60U 3.99U 60U 3.99U 0
V2 0 3 0
V3 4 5
M1 4 1 3 0 NM W=6U L=6U
.END
```

MIN.CIR

```
.TRAN 20N 2.5U 0.1U
.MODEL NM NMOS(LEVEL=3 VTO=0.7 KP=40E-6 GAMMA=1.1
+ PHI=0.60 PB=0.7 CGSO=3.E-10 CGDO=3.E-10
+ CGBO=5.E-10 RSH=25 CJ=4.4E-4 MJ=0.5 CJSW=4E-10
+ MJSW=0.3 JS=1.E-5 TOX=5E-8 NSUB=1.7E16 TPG=1
+ XJ=6.E-7 LD=3.5E-7 UO=775)
.MODEL PM PMOS(LEVEL=3 VTO=-0.8 KP=12.E-6 GAMMA=.6
+ PHI=0.60 PB=0.6 CGSO=2.5E-10 CGDO=2.5E-10
+ CGBO=5.E-10 RSH=80 CJ=1.5E-4 MJ=0.6 CJSW=4E-10
+ MJSW=0.6 JS=1.E-5 TOX=5E-8 NSUB=5.E15 TPG=1
+ XJ=5.E-7 LD=2.5E-7 UO=250)
.PRINT TRAN I(V3) I(V19) I(V18) I(V2) I(V7) I(V17)
+I(V6) I(V20)
M2 1 2 3 4 PM L=6U W=6U
V1 3 0 5V
V2 2 29
V3 1 0
M4 7 7 8 9 PM L=6U W=6U
M5 10 7 8 11 PM L=6U W=6U
V5 8 0 5V
V6 7 20
V7 10 29
M12 24 24 0 12 NM L=6U W=6U
M13 18 24 0 14 NM L=6U W=6U
M14 15 24 0 16 NM L=6U W=6U
V17 29 15
V18 21 20
I1 0 21 PWL (0.4U 0 0.4001U 20U 0.8U 20U 0.8001U 40U 1.2U 40U
+ 1.2001U 60U 1.6U 60U 1.6001U 0 2.0U 0 2.001U 20U 2.4U 20U )
V19 25 24
V20 20 18
I2 0 5 PWL (1.6U 0 1.6001U 20U 1.7U 0 1.8U 0 1.9U 0
+ 2.0U 0 2.001U 20U 2.4U 20U)
V21 25 5 5V
M1 2 2 3 22 PM L=6U W=6U
.END
```

TSUM.CIR

```
M10 5 1 1 5 P W=3U L=3U
M20 5 1 2 5 P W=3U L=3U
M30 5 1 3 5 P W=3U L=3U
M1 3 8 0 0 N W=7.5U L=3U
M2 2 100 6 0 N W=6U L=3U
M3 5 9 4 5 P W=9U L=3U
M4 4 100 6 5 P W=6U L=3U
M5 7 7 0 0 N W=3U L=3U
M6 100 3 0 0 N W=12U L=3U
M7 5 3 100 5 P W=12U L=3U
VOUT 6 7 0
VIN1 1 10 0
VIN2 1 11 0
VDD 5 0 5.0V
VP 9 0 2.43V
VN 8 0 1.61V
I1 10 0 PWL (0.4U 0 0.4001U 20U 0.8U 20U 0.8001U 40U 1.2U 40U
+ 1.2001U 60U 1.6U 60U 1.6001U 0 2.0U 0 2.001U 20U 2.4U 20U
+ 2.4001U 40U 2.8U 40U 2.8001U 60U 3.2U 60U 3.2001U 0 3.6U 0
+ 3.6001U 20U 4.0U 20U
+ 4.001U 40U 4.4U 40U 4.4001U 60U 4.8U 60U 4.8001U 0 5.2U 0
+ 5.2001U 20U 5.6U 20U 5.6001U 40U 6.0U 40U 6.0001U 60U 6.4U 60U)
I2 11 0 PWL (1.6U 0 1.6001U 20U 3.2U 20U 3.2001U 40U 4.8U 40U
+ 4.8001U 60U 6.4U 60U)
.MODEL N NMOS (LEVEL=3 VTO=0.7 KP=40E-6 GAMMA=1.1 PHI=0.6
+ PB=0.7 CGSO=3E-10 CGDO=3E-10 CGBO=5E-10 RSH=25 CJ=4.4E-4 MJ=0.5
+ CJSW=4E-10 MJSW=0.3 JS=1E-5 TOX=5E-8 NSUB=1.7E16
+ TPG=1 XJ=6E-7 LD=3.5E-7 UO=775)
.MODEL P PMOS (LEVEL=3 VTO=-0.8 KP=12E-6 GAMMA=0.6 PHI=0.6
+ PB=0.7 CGSO=2.5E-10 CGDO=2.5E-10 CGBO=5E-10
+ RSH=80 CJ=1.5E-4 MJ=0.6 CJSW=4E-10 MJSW=0.6 JS=1E-5
+ TOX=5E-8 NSUB=5E15 TPG=1 XJ=5E-7 LD=2.5E-7 UO=250)
.OPTION LIMPTS=500
.TRAN 0.1US 6.4US
.PRINT TRAN I(VIN1) I(VIN2) I(VOUT)
.END
```

CYCLE.CIR

```
.TRAN 0.1U 6U
.PRINT TRAN I(V8) I(V3)
.MODEL NM NMOS (LEVEL=3 VTO=0.7 KP=40E-6 GAMMA=1.1 PHI=0.6
+ PB=0.7 CGSO=3E-10 CGDO=3E-10 CGBO=5E-10 RSH=25 CJ=4.4E-4 MJ=0.5
+ CJSW=4E-10 MJSW=0.3 JS=1E-5 TOX=5E-8 NSUB=1.7E16 TPG=1
+ XJ=6E-7 LD=3.5E-7 UO=775)
.MODEL PM PMOS (LEVEL=3 VTO=-0.8 KP=12E-6 GAMMA=0.6 PHI=0.6
+ PB=0.7 CGSO=2.5E-10 CGDO=2.5E-10 CGBO=5E-10
+ RSH=80 CJ=1.5E-4 MJ=0.6 CJSW=4E-10 MJSW=0.6 JS=1E-5
+ TOX=5E-8 NSUB=5E15 TPG=1 XJ=5E-7 LD=2.5E-7 UO=250)
M3 38 8 7 4 PM L=3U W=3U
M4 9 8 7 5 PM L=3U W=3U
M5 1 2 7 6 PM L=3U W=6U
M6 16 20 1 7 PM L=3U W=6U
V1 7 0 5V
V2 2 0 2.43V
V3 25 0
M7 16 20 10 0 NM L=3U W=3U
M9 10 11 0 13 NM L=3U W=6U
M10 38 11 0 12 NM L=4U W=6U
M11 16 16 0 14 NM L=3U W=3U
M12 22 16 0 15 NM L=3U W=3U
V4 11 0 1.61V
V5 9 16
M14 18 22 68 21 PM L=3U W=3U
V8 18 17
V9 68 0 5V
M19 20 37 33 34 PM L=3U W=3U
M20 20 37 0 35 NM L=3U W=3U
V12 33 0 5V
M21 37 38 39 40 PM L=3U W=3U
M22 37 38 0 41 NM L=3U W=3U
V13 39 0 5V
I1 8 19 PWL 0 0 1U 0 1U 20U 2U 20U 2U 40U 3U 40U 3U 60U
+ 3.99U 60U 3.99U 0
M23 19 25 25 0 NM L=3U W=3U
M25 17 17 0 0 NM L=3U W=3U
M13 22 22 68 68 PM L=3U W=3U
```

```
M2 8 8 7 3 PM L=3U W=3U
.END
```

Bibliography

[1] K. C. Smith. Multiple-valued logic - a tutorial and appreciation. *IEEE Computer*, 21:17–27, April, 1988.

[2] Tich T. Dao. Multiple-valued I^2L, applications and extensions. *Computer Science and Multiple-Valued Logic, Theory and application, North Holland, D. C. Rine, Ed*, 1984.

[3] Stanley L. Hurst. Multiple-valued Logic - It's status and its future. *IEEE Transaction on Computers*, C-33(12):1160–1179, December, 1984.

[4] Atul K. Jain. Multiple-Valued Logic Design in Current-Mode CMOS. *Ph.D Dissertation, University of Saskatchewan, Saskatoon, Canada*, August, 1993.

[5] D. Etiemble and M. Israel. Comparison of binary and multi-valued ICs according to VLSI criteria. *IEEE Computer*, 21:28–42, April, 1988.

[6] K. W. Current. Current-Mode CMOS Multiple-Valued Logic Circuits. *IEEE Journal of Solid State Circuits*, 29(2):95–107, February, 1994.

[7] P. P. Trimulai. Multiple Valued Logic PLAs. *Ph.D Dissertation, Northwestern University*, June, 1989.

[8] K. C. Smith. The prospects for Multivalued Logic: A Technology and Application View. *IEEE Transactions on Computers*, C-30(9):619–634, September, 1981.

[9] J. T. Butler. Multiple Valued Logic. *IEEE Computer*, pages 13–15, April, 1988.

[10] M. Stark. Two bits per cell ROM. *Proceedings of COMPCON, San francisco*, pages 209–212, 1981.

[11] M. Stark. N bits per cell ROM. *Computer Science and Multiple-Valued Logic, Theory and Application, North Holland, D. C. Rine, Ed*, pages 538–554, 1984.

[12] David A. Rich. A survey of multi-valued memories. *IEEE Transactions on Computers*, C-35:99–106, February, 1986.

[13] C. R. Edwards. I^2L threshold circuits for binary-quaternary encoding and decoding. *International Journal of Electronics*, 44(4):445–448, 1978.

[14] S. Kawahito, M. Kameyama, T. Higachi, and H. Yamada. A 32×32 bit multiplier using multiple-valued MOS Current-mode circuits. *IEEE Journal of Solid State Circuits*, 23(1):124–132, February, 1988.

[15] T. Sasao. Multiple-valued decomposition of generalized boolean functions and the complexity of programmable logic arrays. *IEEE Transactions on Computers*, C-30:635–642, 1981.

[16] A. Druzeta, Z. G. Vranesic, and A. S. Sedra. A higher-radix technique for fault detection in many valued multi-threshold networks. *IEEE Transactions on Computers*, C-27:1070–1073, November, 1978.

[17] T. Sasao. Input variable assignment and output phase optimization of PLA's. *IEEE Transactions on Computers*, C-33:874–894, October, 1984.

[18] K. C. Smith and P. Glenn Gulak. Prospects for Multiple-Valued Integrated Circuits. *IEICE Trans. Electron*, E76-C(3):175–188, March, 1993.

[19] Z. Vranesic. Application and scope of multiple-valued LSI. *Proc. 1981 Spring COMPCON conf., IEEE Computer Society Press*, pages 213–216, February, 1981.

[20] A. Heung and H. T. Mouftah. Depletion/Enhancement CMOS for lower power family of three-valued logic circuits. *IEEE Journal of Solid State Circuits*, SC-20:609–615, April, 1985.

[21] J. T. Butler and H. G. Kerkhoff. Multiple-Valued CCD circuits. *IEEE Transactions on Computers*, 21:58–69, April, 1988.

[22] M. H. Abd-El-Barr. CMOS quaternary logic encoder-decoder circuits. *International Journal of Electronics*, 71:279–295, February, 1991.

[23] M. H. Abd-El-Barr, Z. G. Vranesic, and S. G. Zaky. Algorithmic synthesis of MVL functions for CCD implementation. *IEEE Transactions on Computers*, C-40:977–986, August, 1991.

[24] Z. G. Vranesic and K. M. Waliuzzaman. Functional transformation in simplification of multivalued switching functions. *IEEE Transactions on Computers*, pages 102–105, January, 1972.

[25] Gamal H. Abdel-Hamid and Mostafa H. Abd-El-Barr. Decomposition-based synthesis of multiple-valued functions for threshold logic network realization. 24^{th} *International Symposium on Multiple-Valued Logic*, pages 58–64, May, 1994.

[26] Tsumoto Sasao. On the optimal design of multiple-valued PLAs. *IEEE Transactions on Computers*, C-38:582–592, April, 1989.

[27] Richard.L. Rudell and Alberto Sangiovanni-vincentelli. Multiple-valued minimization for PLA optimization. *IEEE Transactions on Computer Aided Design*, CAD-6:727–750, September, 1987.

[28] P. Trimulai and J. T. Butler. On the realization of multiple-valued logic functions using CCD PLA's. *Proceedings of the* 18^{th} *International Symposium on Multiple-Valued Logic*, pages 226–236, May, 1988.

[29] G. W. Dueck and D. M. Miller. A 4-valued PLA using the MODSUM. *Proceedings of the* 17^{th} *International Symposium on Multiple-Valued Logic*, pages 221–227, May, 1987.

[30] M. H. Abd-El-Barr. Programmable synthesis of multi-valued multi-threshold functions for implementation using charge-coupled devices. *International Journal of Electronics*, 73(2):345–370, February, 1992.

[31] Atul K. Jain, R. J. Bolton, and M. H. Abd-El-Barr. CMOS Multiple-Valued Logic Design -Part 1:circuit implementation. *IEEE Transactions on Circuits and Systems*, 40(8):503–514, August, 1993.

[32] Y. Hata, F. Miyawaki, and K. Yamato. Optimal output assignment and the maximum number of implicants needed to cover multiple-valued logic functions. *Proceedings of the* 22^{nd} *International Symposium on Multiple-Valued Logic*, pages 389–395, May, 1992.

[33] J. K. Lee and J. T. Butler. Tabular methods for the design of multiple-valued circuits. *Proceedings of the* 13^{th} *International Symposium on Multiple-Valued Logic*, pages 162–170, May, 1983.

[34] H. G. Kerkhoff and H. A. J. Robroek. The logic design of multiple-valued logic functions using charge-coupled devices. *Proceedings of the* 12^{th} *International Symposium on Multiple-Valued Logic*, pages 35–44, May, 1982.

[35] P. Trimulai and J. T. Butler. Minimization algorithms for multiple-valued programmable logic arrays. *IEEE Transactions on Computers*, 40:167–177, February, 1991.

[36] G. Pomper and J. R. Armstrong. Representation of multi-valued functions using the direct cover method. *IEEE Transactions on Computers*, C-30:674–679, September, 1981.

[37] G. W. Dueck and D. M. Miller. A direct cover MVL minimization using the truncated sum. *Proceedings of the* 17^{th} *International Symposium on Multiple-Valued Logic*, pages 221–226, May, 1987.

[38] J. M. Yurchak and J. T. Butler. HAMLET- An expression compiler/optimizer for the implementation of heuristics to minimize multiple-valued programmable logic arrays. *Proceedings of the* 20_{th} *International Symposium on Multiple-Valued Logic*, pages 144–152, May, 1990.

[39] K. W. Current. Current-Mode CMOS Multiple-Valued Logic Circuits. *IEEE Journal of Solid State Circuits*, 29(2):95–107, February, 1994.

[40] F. J. Pelayo, A. Prieto, A. Lloris, and J. Ortega. CMOS current-mode multi-valued PLAs. *IEEE Transactions on Circuits and Systems*, 38(4):434–441, April, 1991.

[41] Daniel Etiemble. On the performance of Multi-valued Integrated Circuits: Past, Present and Future. *Proceedings of the* 22^{nd} *International Symposium on Multiple-Valued Logic*, pages 156–164, May, 1992.

[42] E. L. Post. Introduction to a general theory of elementary proposition. *American Journal of Mathematics*, 43:163–185, 1921.

[43] C. Michael Allen and Donald D. Givone. The Allen-Givone implementation oriented algebra. *Computer Science and Multiple-Valued Logic, Theory and application, North Holland, D. C. Rine Ed.*, pages 268–288, 1984.

[44] Z. G. Vranesic, E. S. Lee, and K. C. Smith. A many-valued algebra for switching systems. *IEEE Transactions on Computers*, C-19:964–971, October, 1970.

[45] S. Onneweer, H. Kerkhoff, and J. T. Butler. Structural computer-aided design of current-mode cmos logic circuits. *Proceedings of the* 18^{th} *International Symposium on Multiple-Valued Logic*, pages 21–30, May, 1988.

[46] M. Davio and J. P. Deschamps. Synthesis of discrete functions using I^2L technology. *IEEE Transactions on Computers*, C-30:653–661, September, 1981.

[47] Sami B. Abugharbieh and Samuel C. Lee. A fast algorithm for the disjunctive decomposition of m-valued functions Part: 1. *Proceedings of the* 23^{rd} *International Symposium on Multiple-Valued Logic*, pages 118–125, May, 1993.

[48] V. Shen, A. McKellar, and P. Weiner. A fast algorithm for the disjunctive decomposition of switching functions. *IEEE Transactions on Computers*, C-20:304–309, March, 1971.

[49] T. Sasao. Multiple-Valued Logic and optimization of Programmable Logic Arrays. *IEEE Computer*, 21:71–80, April, 1988.

[50] K. Vijayan Asari and C. Eswaran. An optimization technique for the design of multiple valued PLA's. *IEEE Transactions on Computers*, 43:118–122, January, 1994.

Vitae

- Muhammad Nayyar Hasan
- Born in 1965 at Karachi, Pakistan.
- Received Bachelor of Engineering (**B.E.**) degree in Computer Systems Engineering from N.E.D. University of Engineering and Technology, Karachi, Pakistan in 1990.
- Worked as Assistant Manager in Pakistan Machine Tool Factory (Pvt) Ltd (Ministry of Production, Govt of Pakistan) from 1990 to 1992.
- Received Master of Science (**M.S.**) degree in Computer Engineering from KFUPM, Saudi Arabia in 1995.